CHALMERS
UNIVERSITY OF TECHNOLOGY

Time-optimal control of large-scale systems of systems using compositional optimization

Downloaded from: https://research.chalmers.se, 2023-05-04 23:09 UTC

Citation for the original published paper (version of record):
Hagebring, F., Lennartson, B. (2019). Time-optimal control of large-scale systems of systems using compositional optimization. Discrete Event Dynamic Systems: Theory and Applications, 29(3): 411-443. http://dx.doi.org/10.1007/s10626-019-00290-0

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology.
It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004.
research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Discrete Event Dynamic Systems (2019) 29:411–443
https://doi.org/10.1007/s10626-019-00290-0

Time-optimal control of large-scale systems of systems using compositional optimization

Fredrik Hagebring[1] · **Bengt Lennartson**[1]

Received: 5 September 2018 / Accepted: 26 July 2019 / Published online: 30 August 2019
© The Author(s) 2019

Abstract

Optimization of industrial processes such as manufacturing cells can have great impact on their performance. Finding optimal solutions to these large-scale systems is, however, a complex problem. They typically include multiple subsystems, and the search space generally grows exponentially with each subsystem. In previous work we proposed *Compositional Optimization* as a method to solve these type of problems. This integrates optimization with techniques from compositional supervisory control, dividing the optimization into separate sub-problems. The main purpose is to mitigate the *state explosion problem*, but a bonus is that the individual sub-problems can be solved using parallel computation, making the method even more scalable. This paper further improves on compositional optimization with a novel synchronization method, called *partial time-weighted synchronization* (PTWS), that is specifically designed for time-optimal control of asynchronous systems. The benefit is its ability to combine the behaviour of asynchronous subsystems without introducing additional states or transitions. The method also reduces the search space further by integrating an optimization heuristic that removes many non-optimal or redundant solutions already during synchronization. Results in this paper show that compositional optimization efficiently generates global optimal solutions to large-scale realistic optimization problems, too big to solve when based on traditional monolithic models. It is also shown that the introduction of PTWS drastically decreases the total search space of the optimization compared to previous work.

Keywords Large-scale optimization · Discrete event systems · State explosion problem · Time-optimal control · Compositional optimization

This article belongs to the Topical Collection: *Smart Manufacturing - A New DES Frontier*
Guest Editors: Rong Su and Bengt Lennartson

This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation

✉ Fredrik Hagebring
fredrik.hagebring@chalmers.se

[1] Division of Systems and Control, Department of Electrical Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden

1 Introduction

Autonomous systems are becoming more and more important in society and especially in industry. This applies also to manufacturing industry, where the level of automation is continuously increasing. The goal is to enable systems, also referred to as *plants*, to take independent decisions within an often unstructured and complex environment, in order to reduce the need of human intervention. To reach this goal, new and fast methods for large-scale optimization that can incorporate all available information must be developed.

Modelling manufacturing systems as *discrete event systems* (DES) (Cassandras and Lafortune 2008) allows for verification and synthesis using formal methods, such as *supervisory control theory* (SCT), first defined by Ramadge and Wonham (1987) and Ramadge and Wonham (1989). However, verification and control of discrete systems are related to combinatorial optimization, and the algorithms suffer from the well-known *state explosion problem*, also called the *curse of dimensionality* (Gass and Fu 2013; Valmari 1998). Wong and Wonham (1998) showed that this can be mitigated to some extent by modular or compositional algorithms when the system is separable into subsystems (system of systems). It has been shown in later work by Flordal and Malik (2009) and Mohajerani et al. (2014) among others that compositional supervisory control can efficiently synthesize controllers for large-scale systems.

The downside of SCT for autonomous systems is that most work focuses on maximally permissive control synthesis for a given set of specifications (Cassandras and Lafortune 2008, chap. 3). The controller should ensure that something bad never happens. This is useful when the plant is operated by an external controller or human operator. An autonomous system needs a controller that can take *good decisions*, in order to eventually let the system reach a predefined goal state. This requires that the model is extended with a cost function that defines the notion of *good*. The controller should then reach the goal as cheap as possible, which constitutes an optimization problem.

A wide range of efficient methods for solving these specific type optimization problems have been explored over the years. There are a using a wide range of different optimization techniques. For further references we recommend Passino and Antsaklis (1989), Brandin and Wonham (1994), Huang and Kumar (2008), Kobetski and Fabian (2009) and Hagebring et al. (2016). Many of them have been proven efficient with respect to computational complexity and typically scales polynomially with the size of the system. Moreover, the type of problems that is addressed can typically be perceived to relate directly to other large field of optimization research such as planning and scheduling. For example, MDP theory, which is the most basic modeling tool for stochastic scheduling, is claimed in the textbook by Cassandras and Lafortune (2008) as a formal DES framework. Regardless of the modelling tool or the solution method, all suffer from the state explosion problem. The problem of addressing this problem have of course been investigated in a large number of publications. For further references we recommend Powell (2007), Cao X (2007), Bertsekas and Tsitsiklis (1996) and Bertsekas (2005).

However, to the best of our knowledge, none of these methods offers a generalized compositional optimization approach, which means that they all have to consider the full search space of the monolithic system. It is not enough with methods that scales polynomially with the size of the system if the system itself scales exponentially with the number and size of its subsystems. The problem with modular or compositional methods in optimization is that there is not enough information locally to fully optimize the subsystem and still guarantee a global optimal solution. In recent years, related work has been presented by other groups on modular or compositional methods. Particularly interesting are the works done by Hill

and Lafortune (2016, 2017), Su (2012a) and Ware and Su (2017). The latter are closely related to the work presented in this paper, where they propose a compositional method for synthesis of a time-optimal controller. The techniques are, however, either restrictive in their reduction of the subsystems or offer only approximative solutions. We claim that the work presented in this paper offers a stronger mitigation of the state explosion, while still generating global optimal solutions.

In Hagebring and Lennartson (2018) we presented a general formulation of a *compositional optimization* method for system of systems, hereinafter called *CompOpt*. This method integrates techniques from compositional supervisory control with traditional graph based search algorithms. Its strength comes from the ability to reduce the state space of each subsystem individually by exploiting their local behavior, mitigating the state explosion that otherwise would occur during synchronization. It was shown that CompOpt drastically reduced the search space during the optimization of a realistic large-scale example and, hence, improved the computational complexity. Dividing the optimization into multiple independent sub-problems also allows for a parallel computation of their solutions. The scalability gained by this is considered an important property of CompOpt. Yet, the added benefit of parallelization has not been included in the evaluation of this paper. Instead it is left to be investigated in future research.

There are several industrial applications where an optimization using CompOpt may be beneficial. This paper provides examples both from logistics, in the motivating example of Section 3, and manufacturing industry, in the large-scale examples of Section 6. The general formulation of CompOpt does, however, enable it to optimize any system of systems as long as these can be modelled using weighted automata, such as in these examples. These type of systems can be found in a wide range of applications and are in no way restricted to only the traditional areas of industrial automation.

One of the main limitation of the previous implementation of CompOpt was the decrease in computational performance when dealing with time-optimal control. This was caused by the non-trivial task of modelling the parallel execution of subsystems. Yet, this type of time-weighted systems is one of the most common applications, e.g. minimizing cycle time of a production cell. Optimization of industrial processes usually consider time as the main cost when improving productivity. Similarly to Ware and Su (2017), the previous implementation of CompOpt used *tick automata* (Gruber et al. 2005) during the synchronization of time-weighted systems. The problem with this is that the technique includes a discretization of the time line, which increases the search space and reduces the overall efficiency. There exists a wide variety of other modelling tools specifically designed for time-weighted systems, the most well known probably being *Timed Automata* (Alur and Dill 1994). Another modelling technique, called *time-weighted automata*, proposed by Su et al. (2012b), is quite similar to the *weighted automata* used in this paper. However, all these modelling techniques add additional information and restrictions to the models. This is required to explicitly represent the full synchronous composition of the time-weighted system. Fortunately, we show in this paper that CompOpt does not require a full synchronous composition.

In this paper we improve on CompOpt by proposing a novel and efficient synchronization method for time-weighted systems, called *partial time-weighted synchronization* (PTWS). PTWS is able to synchronize the parallel behaviour of time-weighted subsystems without adding any additional states or transitions to their models. The key to this method is the integration of an optimization heuristic that, similarly to the local optimization, reduces the state space of the synchronous composition by removing non-optimal or redundant solutions, while maintaining the global optimal solution. We show in this paper that this further improves the efficiency of CompOpt by strengthening the mitigation of the state explosion

problem. The addition of PTWS does not change the main process of CompOpt, it only extends the method with a more efficient synchronization of the subsystems.

The paper is organized as follows. In Section 2 the basic notation and preliminaries are introduced. A motivating example of a logistics system is presented in Section 3, illustrating the impact of the state explosion problem and the benefit and challenges of using a compositional optimization approach. Section 4 gives an introduction to compositional optimization in general and defines the theory behind CompOpt. Section 5 presents PTWS, the integrated optimization and synchronization method that is the main contribution of this paper. In Section 6 we illustrate the potential of CompOpt and especially highlights the improvements gained by PTWS compared to previous work. Finally, Section 7 concludes the paper.

2 Preliminaries

Discrete event systems are modelled in this paper as non-deterministic finite automata (NFA), defined by a 5-tuple $G = (Q, \Sigma, \rightarrow, q_0, Q_m)$, where Q is a set of states, Σ is a finite set of events, $\rightarrow \subseteq Q \times \Sigma \times Q$ is a transition relation, where $q \xrightarrow{\sigma} q' \in \rightarrow$ denotes the transition from the source state q with the event label σ to the target state q', $q_0 \in Q$ is the initial state and $Q_m \subseteq Q$ is a set of marked states. $\Sigma(q) = \{\sigma \in \Sigma \mid (\exists q' \in Q)\, q \xrightarrow{\sigma} q' \in \rightarrow\}$ is the active set of events in state q.

When an automaton G is executed, a set of sequential transitions occurs. By merging this sequence of transitions, a *path* is generated. The target state of the current transition is then merged with the source state of the next transition. Repeating this n times results in the path

$$\rho = q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \xrightarrow{\sigma_3} \cdots \xrightarrow{\sigma_n} q_n, \tag{1}$$

where each transition $q_i \xrightarrow{\sigma_i} q_{i+1} \in \rightarrow$. For convenience, a path is sometimes considered as the set of sequential transitions that constitutes the path, which enables the use of set theory to reason about paths. Thus, the path ρ defined in Eq. 1 can also be written

$$\rho = \left\{ q_0 \xrightarrow{\sigma_1} q_1, q_1 \xrightarrow{\sigma_2} q_2, \cdots, q_{n-1} \xrightarrow{\sigma_n} q_n \right\}. \tag{2}$$

The set $\text{Paths}(q_i, q_j)$ is the set of all paths ρ in G starting in the state q_i and stopping in the state q_j. The set $\text{Paths}(G)$ is the set of all possible paths available in G. A path $\rho \in \text{Paths}(q_i, q_j)$ is *accepting* if $q_j \in Q_m$ and the set $\text{Paths}(q_i, Q_m)$ denotes the set of all accepting paths starting in q_i. Finally, the *natural projection* (Cassandras and Lafortune 2008, chap. 2) of a path ρ from a set of events Σ to the set Ω, is defined as

$$P_{\Sigma \rightarrow \Omega}(\rho) = \left\{ q_i \xrightarrow{\sigma} q_j \in \rho \mid \sigma \in \Omega \right\}. \tag{3}$$

A state q is *reachable* if there exists a path $\rho \in \text{Paths}(q_0, q)$ going from the initial state q_0 to the state q, and q is *coreachable* if there exists an accepting path $\rho \in \text{Paths}(q, Q_m)$ starting in the state q. States that are coreachable are said to be *non-blocking*, since the system can reach a marked state from these states, the opposite being *blocking* states. An automaton G is said to be *trim* if all states are both reachable and coreachable. The notion of sub-automaton $G' \subseteq G$ means that $Q' \subseteq Q$, $\Sigma' \subseteq \Sigma$, $q'_0 = q_0$, $Q'_m \subseteq Q_m$ and $q_1 \xrightarrow{\sigma} q_2 \in \rightarrow'$ implies $q_1 \xrightarrow{\sigma} q_2 \in \rightarrow$ for all $q_1, q_2 \in Q', \sigma \in \Sigma'$.

When applying compositional synthesis to a system of systems, the events of each subsystem can be divided into *local* and *shared* events, where local events appear only in

one single subsystem, while shared events appear in at least two subsystems. Given a system of systems $\mathbf{G} = \{G_1, \ldots, G_n\}$, the local events of each subsystem $G_i \in \mathbf{G}$ is $\Sigma_i^l = \{\sigma \in \Sigma_i \mid \sigma \notin \Sigma_j, \forall j \in [1, n] \setminus \{i\}\}$. The *shared events* is the complement of the local events, $\Sigma_i^s = \Sigma_i \setminus \Sigma_i^l$. A transition labeled by a shared or a local event is referred to as a shared or a local transition, respectively. A path that only includes local transitions are referred to as a local path.

2.1 Weighted automata

To represent the costs of a system we introduce the notion of *weighted automata*. In contrast to a standard automaton, a weighted automaton is a 6-tuple $G = (Q, \Sigma, \rightarrow, q_0, Q_m, c)$ extended with the cost function $c : (q_1, \sigma, q_2) \rightarrow \mathbb{R}^+$, where $q_1, q_2 \in Q$, $\sigma \in \Sigma$, $q_1 \xrightarrow{\sigma} q_2 \in \rightarrow$ and $\mathbb{R}^+$ denotes the set of positive reals. The function defines a unique cost associated with traversing each transition in the automata. It is also extended to cover paths such that $c(\rho)$ gives the total cost of all transitions in path ρ. To simplify notation, we write weighted transitions as $q_1 \xrightarrow{\langle \sigma, w \rangle} q_2$ where $w = c(q_1, \sigma, q_2)$.

Synchronization between weighted automata is equal to the *synchronous composition* $\|$ for regular automata models, defined in Hoare (1978), with the addition that the maximum from each cost function is included as the weight to the new transitions. That is, if $G = G_1||G_2$ is the synchronous composition of two weighted automata, then the cost function c is defined as

$$c((q_1, q_2), \sigma, (q_1', q_2')) = \begin{cases} \max(c_1(q_1, \sigma, q_1'), c_2(q_2, \sigma, q_2')), & \sigma \in \Sigma_1 \cap \Sigma_2 \\ c_1(q_1, \sigma, q_1'), & \sigma \in \Sigma_1 \setminus \Sigma_2 \\ c_2(q_2, \sigma, q_2'), & \sigma \in \Sigma_2 \setminus \Sigma_1 \end{cases} \quad (4)$$

3 Motivating example

This section provides a motivating example to illustrate the impact of the state explosion problem and the potential benefit and challenges of using a compositional optimization approach. The example depicts a simple logistics system, consisting of two delivery trucks that pick up and deliver packages in separate zones. Every day, there are a list of packages that should be picked up and delivered within there operation area. The objective in this example is to deliver all packages as quickly as possible, that is, the goal is to minimize the time when the last truck returns to the warehouse in the afternoon. This motivating example might not really depicts the optimization of a large-scale system of systems, but it is in fact already large enough for the purpose of this illustration. The system is illustrated in Fig. 1.

The figure shows the two trucks and there respective zone. In the center of the area there is a warehouse, which is where the trucks must start and end each day. The figure also includes an example of a scenario where nine packages should be picked up and delivered during the day. The pick up and delivery location of these packages are marked with dots on the map, where the labels $i\,P$ and $i\,D$ represent the pick up and delivery locations of package i respectively. Some packages should be picked up in one zone but delivered in another. In these cases the truck that picks up the package has to bring it back to the central warehouse where it can be moved over to the delivery truck. These type of switches between the trucks are assumed to occur only once a day. The weights to be considered by the cost function should in this case represent the time it takes to perform each task. The tasks include the pick up and delivery of packages, as well as the travel between these locations.

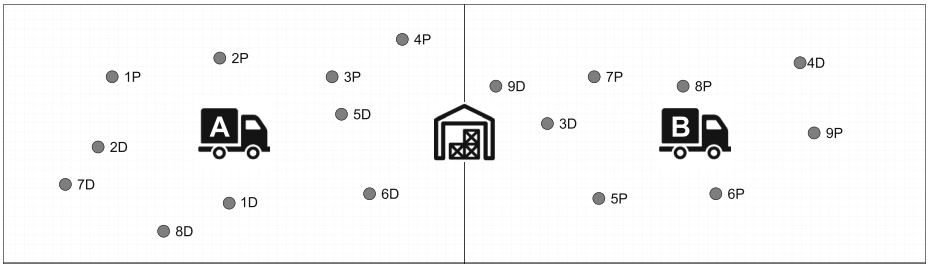


Fig. 1 Illustration of a simple logistics system, consisting of two delivery trucks A and B, operating in adjacent neighbourhoods, that should pick up and deliver a total of nine packages. The pick up and delivery location of a package i is marked iP and iD respectively

The physical position of each truck, can be modelled as a strongly connected graph, where nodes represent the locations of the warehouse and the pick up/delivery tasks, while the edges represent the travel in between. The actual pick up and delivery operations can be modelled as self loops in the nodes of the graph, indicating that a task is performed but the physical location does not change. In favor of readability, a reduced example where truck A only have to pick up and deliver package 1 and pick up package 2 is modelled using a simple automaton in Fig. 2. The markings of the transitions are: (i) the self loops marked by $\langle x \rangle$ illustrating the different operations that can be performed in each location, including $\langle W \rangle$, which represents that the trucks switch packages at the central warehouse, and (ii) the edges between different locations marked by $\langle x, y \rangle$ representing the travel between two locations x and y. The central warehouse is marked green to illustrate that this is the desired goal state, the *accepting* state.

In addition to a model of the *possible* behavior, there are of course also models of the *desired* behavior. These are specified in Fig. 3. The specification in Fig. 3a is applied to all packages that should be picked up and delivered by the same truck. It specifies that the package has to be picked up and then delivered to its final delivery location exactly once.

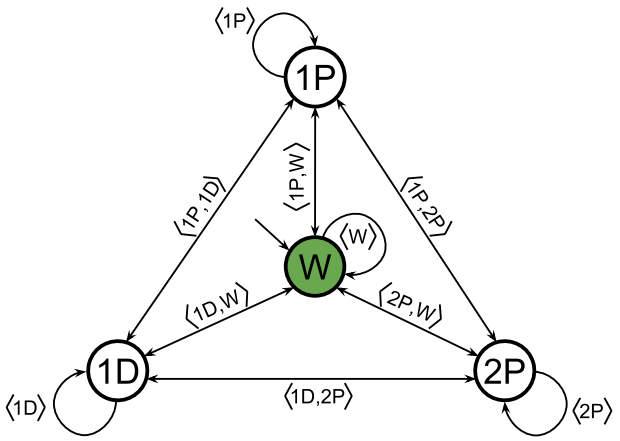


Fig. 2 An automata model of the possible behavior of truck A, when assigned the tasks to pick up packages 1, 2 and deliver package 1. States represent the physical locations, while edges represent operations in these locations and travel in between. The state W represents the central warehouse, which is both the initial and the accepting state of the model

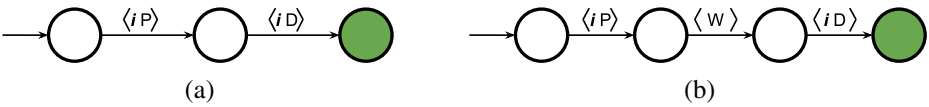


Fig. 3 Generalized models of individual specifications for the route of each package. **a** applies to packages that is picked up and delivered by the same truck, **b** applies to packages that should be picked up by one truck and delivered by another

The specification in Fig. 3b is similar to Fig. 3a but should be applied whenever a package is to be picked up in one zone by truck X and delivered to another zone by truck Y. It is then required that the package is switched from one truck to the other in the central warehouse. Individual specifications like these have to be included for each package.

To evaluate the example, the scenario from Fig. 1 is modelled as a system of systems, using plant models for each truck and specifications for each package to represent the subsystems, such as shown in Figs. 2 and 3. Any optimization applied using a monolithic approach would have to consider a search space spanning the complete synchronized behavior of all subsystems. This is true regardless of the optimization paradigm that is used. Advanced paradigms, such as MILP, CP, might be able to perform clever pruning of the search space in an early stage, but initially all possible combinations of states and transitions have to be considered. This is a potential problem since the size of the search space grows exponentially, due to the state explosion problem. The search space of the simple example shown here includes 342,144 states and 6,329,115 transitions, representing the synchronous composition of all subsystems.

When solving the same example using CompOpt, the optimization problem is partitioned into multiple sub-problems but the sum of states in the search spaces of all sub-problems combined only adds up to 16,396 states. The reason that CompOpt is able to perform so much better than the monolithic approach is the ability to reduce the subsystems even before they are synchronized. The full search space is never computed, no unnecessary states have to be pruned away or evaluated. It is worth noting that CompOpt only represents one specific compositional approach, which most certainly can be further enhanced, but the purpose of this example is just to illustrate that there is much to gain from the ability to optimize systems of systems compositionally.

One could argue that there might exist more efficient models of this system than what is shown here. To a human it is for example obvious that the trucks can be partially optimized individually, since they drive in separate areas, have separate lists of tasks and so on. It is, however, not obvious exactly how this problem can be partitioned since there still exist dependencies between the trucks. Without digging into the details of exactly which tasks that can be considered local, there is no way to partition this problem manually. One benefit of using CompOpt is that it reduces the need of *smart* manual partitioning of the optimization problem, since it already exploits the local behavior of the subsystems.

4 Compositional optimization

Compositional optimization is in this paper proposed as an appealing approach for the optimization of large-scale system of systems. The reason is that it potentially can reduce the state explosion problem, which otherwise occur during the synchronization of these systems of systems. The basic concept behind compositional optimization is to find a global optimal solution to a system of systems by combining the subsystems compositionally into larger

and larger models, while performing local (partial) optimization on each model individually. In this case *global optimality* refers to a global plan or schedule that, when executed, lets the whole system, including all subsystems, transcend from the initial state to a predefined goal state, while minimizing the total cost of performed tasks.

In this paper we present CompOpt, a method for compositional optimization of discrete event systems of systems. In CompOpt, the system of systems is modelled as a set of weighted automata, representing each subsystem and specification individually. These weighted automata include necessary information about the costs and the constraints of the optimization. The cost function is given by the weights on the transition, activating a specific transition is related to a certain cost. The constraints are defined by the structure of the included models, such as their initial state, available transitions, the set of marked states as well as the dependencies between the subsystems identified by shared transitions. Based on this, the objective of CompOpt can be defined as: given a set of weighted automata $G_1, G_2, \ldots, G_n$, find a sequence of transitions, a path, that lets all subsystems transcend from their initial state to a marked state while minimizing the total cost of the activated transitions. That is, let $G = (Q, \Sigma, \rightarrow, q_0, Q_m, c)$ be the monolithic model of the system, represented by the synchronous composition $G = G_1 \parallel G_2 \parallel \ldots \parallel G_n$. The global optimal solution is then a path ρ^*, such that

$$\rho^* = \underset{\rho \in \text{Paths}(q_0, Q_m)}{\arg\min} \; c(\rho). \tag{5}$$

The basic idea in CompOpt is: (i) to use a local optimization algorithm to compute minimal reductions of each subsystem, called *locally optimal reductions*, (ii) synchronize a subset of those locally optimal reductions incrementally to fewer but larger components that include an increasingly larger part of the full system behaviour, (iii) iterate steps (i)-(ii) to further reduce and combine the larger components until only one component remains. The final component will, by construction, be the global optimal solution to the system. In this way the solution is found without considering the full monolithic model at any step.

The key is the local optimization that enables a reduction of each subsystem individually, while still maintaining global optimality. This optimization does, however, suffer from a major limitation that restricts the potential of the approach. In a system of systems, there are typically dependencies between the subsystems. In CompOpt this is caused by shared events between the subsystems. There is no guarantee that the quickest or cheapest sequence of transitions in an individual subsystem is part of the quickest or cheapest sequence of transitions for the whole system. Choosing a specific path in one subsystem can affect other subsystems. In worst case it can block all further actions in another subsystem, making the system unable to reach the goal. In fact, when considering a single subsystem, there is typically not enough information available about these dependencies to prove any unique local path to be optimal for the global system. The local optimization is instead limited to a partial optimization of the subsystems, optimizing only those parts that has no external dependencies. This way it may still reduce the state space of the subsystems and, hence, mitigate the state explosion problem. The main challenge then becomes to identify these independent parts and to reduce these maximally.

To further illustrate the local optimization, consider a small system of systems consisting of two subsystems G_1, G_2. Subsystem G_1 is shown in Fig. 4a, where the marking $\{\sigma, x\}$ of transitions indicates that the transition is activated by the event σ and has the weight x. The only available information of G_2 is that the event a is shared between the two subsystems in some way. How they interact is not revealed. The task at hand is to perform local optimization on G_1. Based on the discussion above we know that the shared behavior

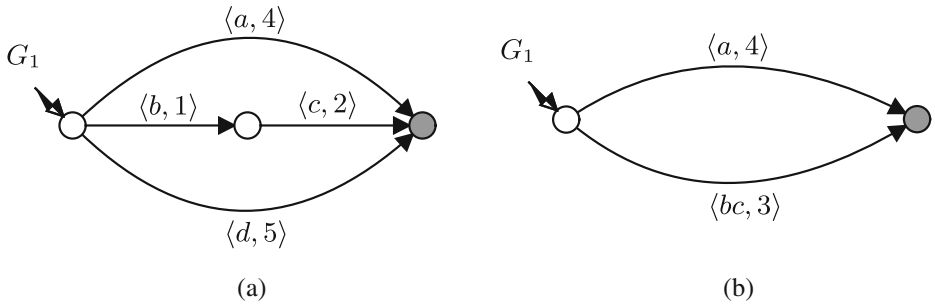


Fig. 4 An illustration of the properties of local optimization. **a** shows a plant model of a subsystem G_1, where it is known that the event a is shared with another subsystem, **b** shows the locally optimal reduction of G_1, where the local transition $\{d, 3\}$ and the sequence $\{b, 1\}, \{c, 2\}$ has been merged into an abstraction $\{bc, 3\}$ that represent there sequential execution

has to be preserved. In this case it means that the shared transition over event a has to be maintained. The rest of the behavior can be considered local and can, hence, be optimized without affecting G_2. The local transition over event d is removed since it has a higher cost than the sequence of local events b, c. The remaining sequence of local events is abstracted to a single transition representing there sequential execution. The resulting locally optimal reduction of G_1, denoted G_1', is shown in Fig. 4b.

The formal definition and properties of local optimization and the implementation of a compositional algorithm is explained in detail in the following sections.

4.1 Local optimization of subsystems

Local optimization defines the process of computing a *locally optimal reduction* G' of a subsystem G. That is, $G' \subseteq G$ is a reduction of G including only states and transitions that is required in order to guarantee a global optimal solution of the monolithic system. In contrast to the maximally permissive supervisors that is the focus of SCT, a locally optimal reduction can be seen as a *minimally permissive* or *maximally restrictive* supervisor that satisfies the specification that all potentially optimal behavior should be maintained. In some cases where it is clear from the context, we use G' also to denote a locally optimal reduction that has been abstracted, such as in the example in Fig. 4.

Since the local optimization considers each subsystem individually, this implicitly requires that the reduction does not modify any of the non-blocking shared behavior of the subsystem. For example no shared transitions from which a marked state can be removed. Doing so might cause a sub-optimal solution or even a blocking of another subsystem. What can be reduced using local optimization is redundant local paths. If there exist two local paths $\rho_1, \rho_2 \in \text{Paths}(q_1, q_2)$ between any pair of states $q_1, q_2 \in Q$, they can be considered redundant and the optimization is free to remove the path with highest cost, or either one if their cost is equal. Additionally, all states that are not reachable or coreachable can also be safely removed since they can not be part of any optimal solution. The reduction can be considered locally optimal when it is trim and a maximum of one such local path remains between any two pair of states.

It is worth mentioning that the locally optimal reduction of a given automaton is not always unique. These may be multiple redundant solution to the local optimization. However, it is proven below that any locally optimal reduction satisfy the required properties of global optimality and minimally permissive.

The aforementioned properties can be summarized in the following definition.

Definition 1 Given a weighted automaton G, then $G' \subseteq G$ is a *locally optimal reduction* of G if:

1. G' is trim
2. For all accepting paths $\rho \in \text{Paths}(q_0, Q_m)$ in G, there exists an accepting path $\rho' \in \text{Paths}(q_0', Q_m')$ in G' such that:
   $P_{\Sigma \to \Sigma \setminus \Sigma^l}(\rho) = P_{\Sigma \to \Sigma \setminus \Sigma^l}(\rho') \wedge c'(\rho') \leq c(\rho)$
3. $|\{\rho \in \text{Paths}(q_1, q_2) \mid \rho \text{ is local}\}| \leq 1, \quad \forall q_1, q_2 \in Q'$

Note that the first part of the conjunction in point 2 of Def. 1 defines, for all accepting paths $\rho \in \text{Paths}(q_0, Q_m)$ in G, the existence of an accepting path $\rho' \in \text{Paths}(q_0', Q_m')$ in G', such that a projection of ρ' that only considers the set of shared events equals corresponding projection of ρ. From this we can infer that the non-blocking shared behavior of G' equals that of G. Point 3 prevents redundant paths in the reduction by allowing a maximum of one local path between any two states q_1, q_2 in the reduction G'.

Based on Def. 1 we can formulate two theorems: Theorem 1 stating that any locally optimal reduction maintains the global optimal solution and Theorem 2 stating that the reduction is minimal.

Theorem 1 *Given a weighted automaton $G = G_1 \parallel G_2$ representing a system of systems, let $G' = G_1' \parallel G_2$ where G_1' is the locally optimal reduction of the subsystem G_1. Then, the global optimal solution of G is also available in G', i.e. for all accepting paths $\rho \in Paths(G)$ there exists an accepting path $\rho' \in Paths(G')$ such that $c'(\rho') \leq c(\rho)$.*

Proof The theorem can then be proven by contradiction using point 2 in Def. 1. Assume that the global optimal solution to G is given by the path

$$\rho^* = \underset{\rho \in \text{Paths}(q_0, Q_m)}{\arg\min} \; c(\rho)$$

and that there exists no accepting path $\rho' \in \text{Paths}(q_0', Q_m')$ in G' such that $c'(\rho') \leq c(\rho^*)$. This implies that at least one transition in the global optimal solution is blocked by the synchronization of the two subsystems in G'.

Since G_2 remains untouched we can infer that the specific sequence that corresponds to the global optimal solution has been removed during the reduction of G_1', i.e. the shared behaviour of the subsystem has changed. This, in turn, requires that the subsystem G_1' fulfills one of two properties: (i) a shared transition has been removed from G_i' blocking the global optimal solution in the synchronization, i.e. there exists an accepting path $\rho \in \text{Paths}(G_1)$ that has no matching accepting path $\rho' \in \text{Paths}(G_1')$ that fulfills $P_{\Sigma \to \Sigma \setminus \Sigma^l}(\rho) = P_{\Sigma \to \Sigma \setminus \Sigma^l}(\rho')$, (ii) at least one local path leading between the shared transitions or marked states in G_1' are sub-optimal. Both of these properties directly violates point 2 of Def. 1, (i) since the definition explicitly requires the existence of an accepting path $\{\rho' \in \text{Paths}(q_0', Q_m') \mid P_{\Sigma \to \Sigma \setminus \Sigma^l}(\rho) = P_{\Sigma \to \Sigma \setminus \Sigma^l}(\rho')\}$, for all $\rho \in \text{Paths}(q_0, Q_m)$, and (ii) since a sub-optimal local path would violate part two of the conjunction, $c(\rho') \leq c(\rho)$. This proves that Theorem 1 holds by definition for any locally optimal reduction. □

Remark It is important to note that we use natural projection to compare only the shared behavior of the reduction, the theorem full behavior of G' does not have to be a projection

of G, i.e. the reduction does not necessarily handle all local transitions equally even if they have the same event, there might exist two local transitions with the same event where one is removed an the other is kept.

Using Theorem 1, we can also induce that the global optimal solution is preserved also when $G'' = G'_1 \parallel G'_2$, where both subsystems has been replaced by their locally optimal reductions. Since the synchronous composition is commutative (Cassandras and Lafortune 2008, chap. 2.3), we know that $G' = G'_1 \parallel G_2 = G_2 \parallel G'_1$. Theorem 1 then states that the global optimal solution of $G' = G_2 \parallel G'_1$ is also available in $G'' = G'_1 \parallel G'_2$. This proves that the global optimal solution is unaffected when combining the subsystems compositionally. However, Theorem 1 does not put any requirements on the reduction, e.g. $G' = G$ fulfills the requirements of Theorem 1 since it maintains the global optimal solution. To verify the optimality of the reduction we need to form Theorem 2.

Theorem 2 *Given a weighted automaton G, let G' be the locally optimal reduction of G. Then, there exists no smaller reduction $G'' \subset G'$ such that G'' is a locally optimal reduction of G'.*

Proof This theorem can be proven by deduction from points 1-3 of Def. 1. As previously mentioned, point 2 of the definition ensures that any locally optimal reduction maintains all non-blocking shared transitions. These can never be reduced. The remaining transitions can be divided into two groups. Firstly, transitions that ends up in blocking states, these are all removed in accordance to point 1 of the definition and, hence, cannot be further reduced. The final group of transitions are the non-blocking local transitions. We can deduct from Def. 1 point 2 that it exist at least one local path between any sequential pair of shared transitions in all accepting paths, and from point 3 that it will exist at most one such local path (since redundancy is not allowed). Hence, no further reduction is possible without violating the definition, which proves that the theorem holds for any locally optimal reduction. □

To compute an abstracted locally optimal reduction of a system G we propose Algorithm 1. The main idea of this algorithm is to initiate a model based only on the non-blocking shared transitions and then connect these shared transitions with the initial state, with each other, and with the marked states using local paths. Finally the algorithm applies an abstraction of the local paths, replacing each sequence of local events with a single event that includes there combined behavior, as previously shown in Fig. 4.

The algorithm first adds all shared transitions of G to G'. Based on these, two sets of states Q_s, Q_t are defined, that represent potential source and target states, respectively, for those local paths that will connect the shared transitions in G'. Set Q_s includes the initial state and the target state of all shared transitions, and set Q_t includes the source state of all shared transitions. The algorithm then considers each source state q_s separately. The shortest local paths from q_s to all other states are calculated using Dijkstra's algorithm (Dijkstra 1959), which basically is a forward search that only include local transitions. For each target state q_t where such a local path is found, the transitions of the path are added to G'. Then, in the same way, the shortest accepting local path from q_s to any marked state is added to G'. When all source states have been processed, the set of marked states Q'_m is generated, completing the locally optimal reduction. Finally, the algorithm performs the subsequent abstraction, where all straight sequences of local transitions are replaced with a single transition, and the original structure is saved in a look-up table τ for later reconstruction.

Algorithm 1 Local optimization: Given a weighted automaton $G = (Q, \Sigma, \rightarrow, q_0, Q_m, c)$ and a set of local events Σ^l, first compute a locally optimal reduction $G' \subseteq G$ and then abstract all local paths into single transitions.

1. Initiate $G' = (Q', \Sigma', \rightarrow', q_0', Q_m', c') := (\{q_0\}, \Sigma, \{\}, q_0, \{\}, c')$ and $\tau := \{\}$, where G' will become the locally optimal reduction and τ is a look-up table including the original structure of abstracted paths, i.e. if $q_1 \xrightarrow{\langle a, w_1 \rangle} q_2 \xrightarrow{\langle b, w_2 \rangle} q_3$ are replaced by $q_1 \xrightarrow{\langle ab, w_1 + w_2 \rangle} q_3$ then $\tau[q_1, q_3] = \{(q_2, a, w_1), (q_3, b, w_2)\}$
2. Trim the original model G
3. For $\{q_i \xrightarrow{\sigma} q_j \in \rightarrow \mid \sigma \in \Sigma \setminus \Sigma^l\}$:
   - i $\rightarrow' := \rightarrow' \cup \{q_i \xrightarrow{\sigma} q_j\}$
   - ii $Q' := Q' \cup \{q_i, q_j\}$
4. $Q_s := \{q_0'\} \cup \{q \in Q' \mid (\exists q_i \in Q') q_i \xrightarrow{\sigma} q \in \rightarrow'\}$
5. $Q_t := \{q \in Q' \mid (\exists q_i \in Q') q \xrightarrow{\sigma} q_i \in \rightarrow'\}$
6. For $q_s \in Q_s$:
   - i Compute the shortest local path $\rho(q) \in \text{Paths}(q_s, q)$ from state q_s to all states $q \in Q$, using Dijkstra's algorithm
   - ii For $q_t \in Q_t$, if $\rho(q_t)$ is defined: Add all transitions of path $\rho(q_t)$ to $\rightarrow'$
   - iii $\rho_m := \underset{q \in Q_m}{\text{argmin}} c(\rho(q))$
   - iv Add all transitions of path ρ_m to $\rightarrow'$
7. $Q_m' := Q_m \cap Q'$
8. For $\{q \in Q' \mid q \neq q_0' \wedge q \notin Q_m'\}$, if the only transitions connected to state q are a single incoming local transition and a single outgoing local transition, i.e. $q_i \xrightarrow{\langle \sigma_i, w_i \rangle} q \xrightarrow{\langle \sigma_j, w_j \rangle} q_j$ and $\sigma_i, \sigma_j \in \Sigma^l$, then:
   - i $Q' := Q' \setminus \{q\}$
   - ii $\rightarrow' := \rightarrow' \cup \{q_i \xrightarrow{\langle \sigma_i \sigma_j, w_i + w_j \rangle} q_j\} \setminus \{q_i \xrightarrow{\sigma_i} q, q \xrightarrow{\sigma_j} q_j\}$
   - iii $\tau[q_i, q_j] := \{(q, \sigma_i, w_i), (q_j, \sigma_j, w_j)\}$
9. Return G', τ

Example Consider the example shown in Fig. 5. Applying Algorithm 1 on G, shown in (a), will return the locally optimal reduction G', shown in (b). Note that the events a and b are shared and local events respectively and the events c and d are local abstractions. The process will include the following steps.

1. Initiate:
   $G' = (Q', \Sigma', \rightarrow', q_0', Q_m', c') := (\{s_0\}, \{a, b\}, \{\}, s_0, \{\}, c'), \tau := \{\}$.
2. G is already trim.
3. Add shared transitions to G':
   $\rightarrow' := \{s_2 \xrightarrow{a,1} s_6, s_3 \xrightarrow{a,2} s_4, s_6 \xrightarrow{a,1} s_8\}$, $Q' := \{s_0\} \cup \{s_2, s_3, s_4, s_6, s_8\}$.

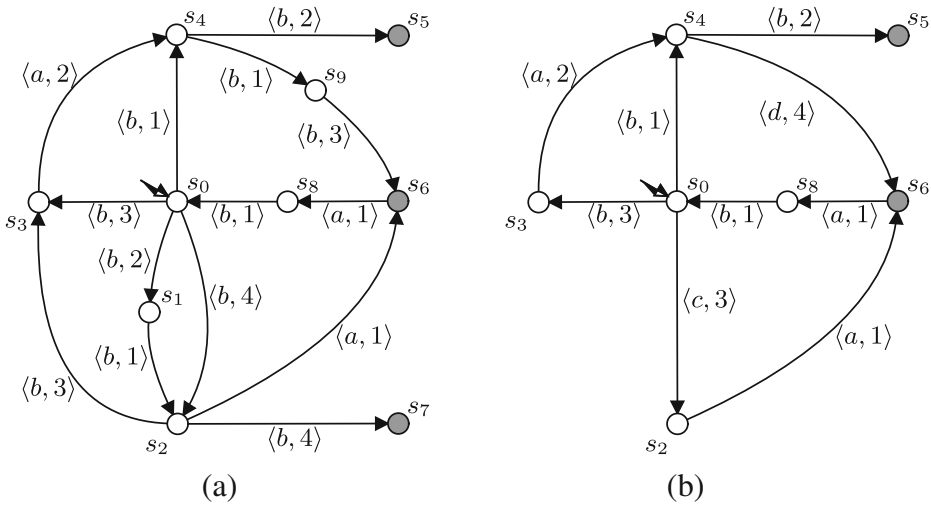


Fig. 5 Local optimization of a weighted automata G, where the events a and b are shared and local events respectively. **a** show the full model G and **b** show the locally optimal reduction G'

4 Since $\rightarrow'$ only includes shared transitions, the source states of potential local paths are:
$Q_s := \{s_0\} \cup \{s_4, s_6, s_8\}$.

5 Similar to step 4, the target states of potential local paths are:
$Q_t := \{s_2, s_3, s_6\}$.

6.1 For $q_s = s_0$, add to $\rightarrow'$, the shortest local path from s_0 to each target state. This adds three paths: $s_0 \xrightarrow{b,2} s_1 \xrightarrow{b,1} s_2$, $s_0 \xrightarrow{b,3} s_3$ and $s_0 \xrightarrow{b,1} s_4 \xrightarrow{b,1} s_9 \xrightarrow{b,3} s_6$. Also add the shortest local path from s_0 to any marked state, which is $s_0 \xrightarrow{b,1} s_4 \xrightarrow{b,2} s_5$.

6.2 For $q_s = s_4$, add to $\rightarrow'$, the shortest local path from s_4 to each target state. This adds only one paths: $s_4 \xrightarrow{b,1} s_9 \xrightarrow{b,3} s_6$, since the other target states are unreachable using local paths. The shortest local accepting path from s_4 is $s_4 \xrightarrow{b,2} s_5$.

6.3 For $q_s = s_6$, this state has no local outgoing transitions and can be neglected.

6.4 For $q_s = s_8$, this state has a single outgoing transition, a local transition leading to the initial state, and, hence, the result from the optimization will be similar to step 6.1 with the addition of the new transition. That is, the following paths will be added: $s_8 \xrightarrow{b,1} s_0 \xrightarrow{b,2} s_1 \xrightarrow{b,1} s_2$, $s_8 \xrightarrow{b,1} s_0 \xrightarrow{b,3} s_3$, $s_8 \xrightarrow{b,1} s_0 \xrightarrow{b,1} s_4 \xrightarrow{b,1} s_9 \xrightarrow{b,3} s_6$ and the accepting path $s_8 \xrightarrow{b,1} s_0 \xrightarrow{b,1} s_4 \xrightarrow{b,2} s_5$.

* *Remark:* Step 6 adds many transitions to $\rightarrow'$ multiple times. This is okay as long as the transition relation is implemented using sets and, hence, only will accept one entry for each unique transition.

7 $Q'_m := \{s_5, s_6, s_7\} \cap \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_8, s_9\}$

8 There are two states in Q' that satisfy the criteria in step 8, $s1$ and s_9. Each of these states are removed. This will add $\tau[s_0, s_2] := s_0 \xrightarrow{b,2} s_1 \xrightarrow{b,1} s_2$ and $\tau[s_4, s_6] := s_4 \xrightarrow{b,1} s_9 \xrightarrow{b,3} s_6$ and the paths will be replaced by the abstracted transitions $s_0 \xrightarrow{c,3} s_2$ and $s_4 \xrightarrow{d,4} s_6$ respectively.

9 This terminates the algorithm and the completed locally optimal reduction G' and the abstraction look-up table τ are returned.

Computational complexity: The complexity of the local optimization is dominated by the computation of the shortest paths from each source state to all other states. This search is done using Dijkstra's algorithm, with a complexity of $\mathcal{O}(V^2)$, in each source state, which gives the local optimization a worst case complexity of $\mathcal{O}(V^3)$ where V is the number of states in the system. Initially we implemented a more complex search algorithm including caching of the partial search results from Dijkstra's algorithm such that no path was required to be searched twice. This looked promising when testing on smaller instances but it scaled poorly with larger instances due to significantly higher memory allocation.

4.2 Compositional optimization

The benefit of the local optimization becomes obvious when integrated with the compositional computation of the global optimal solution, presented in this section. In addition to local optimization of each individual subsystem, the algorithm further mitigates the state explosion by an incremental synchronization of the subsystems, where local optimization is utilized in each step before adding additional subsystems. The incremental process continues until only one final model remains representing the global optimal solution. Comparing this to a monolithic approach, this requires that multiple sub-problems are solved instead of one single optimization, but in return each sub-problem has the potential to be very small compared to the monolithic model. Algorithm 2 presents a complete algorithm for CompOpt.

Algorithm 2 Compositional optimization: Given a set of weighted automata $\mathbf{G} = \{G_1, \ldots, G_n\}$, compute the globally optimal model G'.

1. Initiate $\tau := \{\}$ to be a look up table, such as described in Algorithm 1, including every abstraction performed during all individual reductions
2. For $G_i \in \mathbf{G}$, replace G_i with G_i', τ_i computed by Algorithm 1
3. $\tau := \bigcup \tau_i$
4. While $|\mathbf{G}| > 1$:
   - i Select two arbitrary subsystems $G_i, G_j \in \mathbf{G}$
   - ii $G_{ij} := G_i \parallel' G_j$
   - iii Compute G_{ij}', τ_{ij} using Algorithm 1
   - iv $\tau := \tau \cup \tau_{ij}$
   - v $\mathbf{G} := \mathbf{G} \cup \{G_{ij}'\} \setminus \{G_i, G_j\}$
5. Return G' and τ, where G' is the final remaining component of $\mathbf{G}$

The choice of subsystems in each iteration of step 4.*i* in the algorithm affects the performance, since it determines the amount of reduction that is possible in next step. There are efficient heuristics available to maximize the benefits of the compositional synthesis such as Flordal and Malik (2009). The evaluation of these heuristics has not been included in this work. The systems have instead been synchronized in a predefined sequence in order to isolate the complexity of the method, instead of the efficiency of the heuristics.

Another potential benefit of CompOpt that has not been evaluated in this paper is the inherent ability to compute many of the sub-problems in parallel. It is obvious that the local

optimization of the individual subsystems in step 2 can be performed in parallel. In addition to this, one can also partially parallelize the computations in step 4 by running multiple loops on separate CPU cores, while coordinating the results in a mutual set G. In this case the separate cores can synchronize and optimize different subsystems simultaneously to reduce the total computation time.

Computational complexity: The actual complexity of CompOpt is mainly dependent on the complexity of the shared behavior between the subsystems since this cannot be reduced by the local optimization. Following the Algorithm 2, one can see that Comp-Opt performs local optimization $2n - 1$ times and $n - 1$ synchronizations. Hence, the number of steps in the algorithm grows linearly with the number of subsystems. Sections 4.1 has already shown the local optimization has a complexity that is polynomial in the size of the subsystems and the same is true also for the synchronous composition (Cassandras and Lafortune 2008). However, due to the state explosion problem, the complexity is better described by the growth of the state space during the synchronization of the subsystems. This cannot be guaranteed to always be polynomial. In worst case, when no local behavior exist, the local optimization will be unable to reduce the subsystems at all. This will give an exponential growth of the state space. The complexity can then be simplified to $\mathcal{O}(V^n)$, where V is the number of states in the largest subsystem and n is the number of subsystems. The best case is when all subsystems are disjoint, they can then be reduced to a single transition each through local optimization. In this case the complexity becomes $\mathcal{O}(nV^3)$.

5 Synchronization of time-weighted systems

This section describes the integrated synchronization and optimization operation, called *partial time-weighted synchronization* (PTWS) and denoted $\|'$, which we propose as part of CompOpt to synchronize the behaviour of time-weighted systems.

Time-weighted systems can be considered as a specific class of weighted systems, where the weights connected to the transitions represent their execution time or duration. This means that a time-weighted system is no DES, since the transitions have a duration. In practice, this does not affect the optimization except during the synchronization of the subsystem, where the default synchronous composition no longer can be applied.

To illustrate the implications related to time-weighted systems, consider the automata in Fig. 6, where (a) and (b) are two subsystems G_1, G_2 that run in parallel and (c) represents

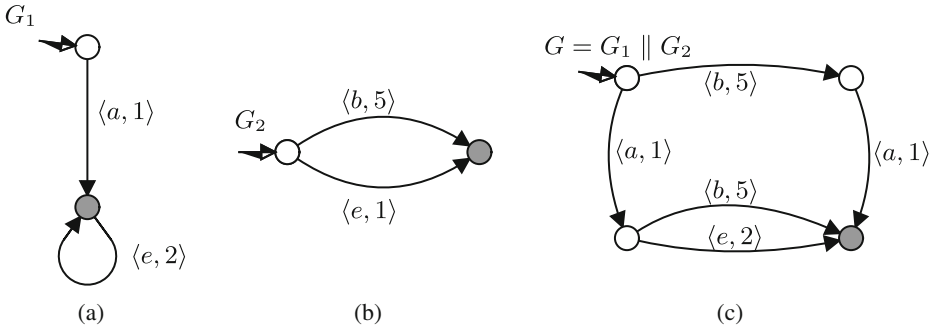


Fig. 6 Example of two subsystems systems G_1, G_2 and their synchronous composition $G = G_1 \parallel G_2$

their synchronous composition G. First assume that the weight associated with each transition represents the energy consumption of the event. Then, the synchronous composition in G is correct. Each event generates an individual energy consumption and the total consumption or cost of a specific path in the model equals the sum of the included weights. If we instead assume that the weights represent the execution time of the transitions, then G no longer models the parallel execution of the subsystems correctly. Consider for example the path in G where event b is executed in the initial state, the sequential representation of the events prevents event a from executing until event b has finished, delaying it for 5 seconds, even though these are strictly local transitions that should be able to run in parallel. In contrast to a parallel execution, the synchronous composition implies that it will be the sum of these execution times that is required to reach the marked state. This shows that the default synchronous composition known from DES is insufficient for the synchronization of time-weighted subsystems when modelled as weighted automata. For a correct synchronization it is required that the execution of each system is tracked individually, since they can, and often will, run in parallel.

In SCT, these systems generally require more complex modelling paradigms, such as *timed automata* (Alur and Dill 1994) or *timed Petri net* (David and Alla 2010). The main problem with these paradigms is that they can not be used to apply many of the efficient verification and synthesis techniques, which are developed for ordinary automata or Petri net models. An alternative approach, which allows the time-weighted system to be modelled using regular automata, is to apply simplifications to the system, e.g. discretization of the time line such as *tick automata* (Gruber et al. 2005). However, we proved in Hagebring and Lennartson (2018) that these are very inefficient, since the discretized time-line resulted in a reduced accuracy as well as a drastically increased state space. This makes simplifications such as tick automata unfit for the use in CompOpt, since the main goal is to mitigate the state explosion problem. For this reason we propose PTWS to support CompOpt in the optimization of time-weighted systems. Once again, remember that the addition of PTWS does not change the main process of CompOpt. The theory presented in Section 4 remains valid, except that, when optimizing time-weighted system, the synchronization in Algorithm 2 is performed using PTWS instead of default synchronous composition.

There are two main properties that distinguish PTWS and makes it especially suitable for CompOpt. Firstly, PTWS solves the previously mentioned implications of time-weighted systems by the introduction of a novel modelling technique, where the execution of the individual subsystems are tracked by extending the state names in the synchronous composition. This enables a correct modelling of parallel time-weighted subsystems using only ordinary weighted automata without applying any simplifications. Secondly, the integration of an optimization heuristic into the synchronization reduces the state explosion of the synchronous composition, which aligns with the main goal of CompOpt.

PTWS is implemented using a single forward search, where the optimization heuristic ensures that the synchronization only expands specific parts of the composition. This disregards many states and transitions that are not needed in order to compute a global optimal solution of the monolithic system. The result is a partial synchronous composition, which still maintains the global optimal solution. The most beneficial effect of this integrated optimization is that it directly mitigates the state explosion problem during every synchronization.

Using a single forward search makes PTWS cheap, both with respect to time and memory, but the heuristic is not sufficiently strong for the result to be a locally optimal reduction. However, similarly to Def. 1 of locally optimal reduction, requirements on the heuristics used by PTWS guarantees that that any PTWS $\|'$ maintains the global optimal solution. The

locally optimal reduction can then be computed subsequent to PTWS, using Algorithm 1, but with the benefit of a much smaller search space.

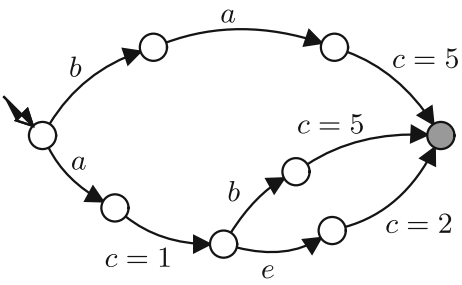
These modelling techniques, using extended state names and the specific optimization heuristic that we propose, are thoroughly described in the following sections.

5.1 Synchronization of time-weighted systems using extended state names

The purpose of using extended state names in the synchronization of time-weighted systems is to avoid more complex modelling paradigm, which otherwise would affect the complexity of the optimization negatively. To better understand the idea behind this technique and the added benefit, consider the model in Fig. 7. This illustrates how a timed automaton can be used to represent a parallel execution of the two subsystems in Fig. 6. Timed automata is one of the best-known paradigm for modelling of temporal properties in discrete event systems. Timed automata are an extension of regular automata where the notion of clocks controls the timing. This can be used to specify the minimum or maximum time that a system may stay in a state before executing a certain event. Note that the example uses a simplified notation of timed automata, where c represents a clock that executes certain transitions after a predefined time. This is used to simulate the duration of event a and b from the subsystems.

The model in Fig. 7 does not represent a full synchronous composition of the two subsystems. Instead, it shows only an example of one possible combination of the two systems, including three accepting paths: (i) 'ba' where a and b are run in parallel, (ii) where a and b are run in sequence and (iii) 'a, e' where G_2 is waiting until event e can be executed both in G_1 and G_2 as a shared transition. The specific choice of transitions to include in this example is in fact based on the heuristic presented below in Section 5.2.

The example above shows that the timed automata model requires one additional state and transition each time the clock is used. This is the type of increased complexity that is avoided using extended state names. Each state name q_i is then extended to a pair (q_i, t_i) where q_i is the original state name and t_i is the time left until all outgoing transitions become activated, i.e. $t_i = 0$ means that the system is ready to perform a new transition while $t_i > 0$ means that it still executes the previous transition. This way of modelling is not meaningful for individual subsystems since the value of t_i is zero in all states and, hence, does not affect the model at all. The extended state names should be utilized only in the synchronized system, where the extended state names, on the form $\langle(q_{1i}, t_{1i}), (q_{2i}, t_{2i})\rangle$, can be used to indicate that one of the subsystems is still waiting for a previous task to finish. Remember that the transitions of a time-weighted system are typically not instantaneous. Instead, the



(a)

Fig. 7 Example of a possible combination of the two asynchronous subsystems G_1, G_2 from Fig. 6, represented by a timed automaton where c is a clock representing the time that the system has waited in a specific state. c is set to zero in each transition

weights of the transitions represent the *time step* required to finish the execution and to reach the next state. The benefit of using extended state names is that the time step of a transition in the synchronized system does not have to equal the time step of the corresponding transition in the subsystem. Instead, the status of each subsystem is tracked in the name of the next state. The extended state names do not affect the final result, which still will be an ordinary weighted automaton.

The method is most easily illustrated using an example, in Fig. 8, showing the same synchronization as in Fig. 7 but with the extended state names instead of a timed automaton. The state names of the two subsystems are extended in Fig. 8a and b. This is done for reference only and is not necessary in the implementation of the algorithm. The synchronized system is given in Fig. 8c. Consider the upper path of G in Fig. 8c, where event b is executed from the initial state for a duration of 0 time unit. The corresponding transition in G_2 has a weight of 5, which means that when G has finished the transition, G_2 still have 5 time units left until it has reached the marked state. This is then indicated in the target state name by setting $\langle (q_{1i}, t_{1i}), (q_{2i}, t_{2i}) \rangle = \langle (s_1, 0), (s_2, 5) \rangle$ saying that G_1 still remains in the initial state, ready to execute an event while G_2 currently transcending towards state s_2 but requires 5 time units until this state is reached. Next, the event a is executed in G and, even if G_1 only requires 1 time unit to execute this event, the synchronized system chose to execute for 5 time units. The reason is of course to allow both G_1 and G_2 to finish there current transition and reach their marked state.

Even in this very small example, one can see that the complexity of this model, in terms of number of states and transitions, is lower than in Fig. 7, which directly affects the resulting computation time in the local optimization.

Just as the timed automata in Fig. 7, the weighted automata in Fig. 8 only represent one possible combination of the two subsystems. This partial synchronization has been applied in these examples since the full synchronous composition requires an infinite number of states when modelled using extended state names. The reason being that the extended state names and the weights of the transitions put strict constraints on the delays between the events.

Consider for example the sequence of event b, a, which in Fig. 8 is represented only by the accepting paths $\langle b, 0 \rangle, \langle a, 5 \rangle$. This specific path represent the case where event b is executed in the initial state, event a is executed at the same time without delay (since the weight of the first transition is zero) and after 5 time units both subsystems have reached a marked state. However, a full synchronous composition of the two system also have to

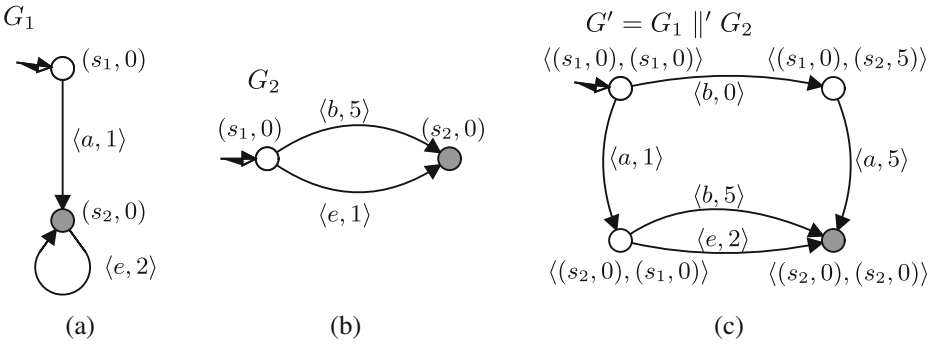


Fig. 8 Compositional modelling of asynchronous systems using extended state names. (a-b) the individual systems G_1, G_2 with extended state names, (c) example of a PTWS $G = G_1 \parallel' G_2$

include all other variations of this event sequence, such as $\langle b, 1\rangle$, $\langle a, 4\rangle$ where event b still executes from the initial state, but an additional delay of 1 time unit is applied before event a can occur. These infinite variations of the event delays makes it impossible to model a full synchronous composition of the two subsystems using this simple method.

Fortunately, CompOpt does not need a full synchronous composition. From Algorithm 2 we remember that CompOpt applies local optimization on the model directly after the synchronization. This means that a partial synchronization is enough as long as the synchronous composition is guaranteed to include at least locally optimal reduction of the full composition. This makes this modelling technique ideal for PTWS when combined with a heuristic that restricts the expansion of the state space in the synchronization. The specific heuristic that we propose is presented in the next section.

5.2 Heuristic for partial time-weighted synchronization

We can see above that integrating an optimization heuristic into PTWS offers clear benefits to CompOpt. However, any heuristic that is used in PTWS has to fulfill two main requirements, it should: (i) restrict the expansion of the synchronization such that a finite composition can be computed, (ii) maintain the global optimal solution of the full system.

The heuristic that is presented in this paper is static in the sense that it does not require information about the previous or future transitions, instead it works only with the information of the outgoing transitions from the current states in the subsystems. The main benefit of this is that the implementation is very memory efficient. This comes at the expense of a less powerful reduction that is not enough to generate a locally optimal reduction directly during the synchronization. Instead, an additional optimization, using Algorithm 1, is required when the synchronization is finished. Experiments were first conducted using a more advanced heuristic that utilized an extensive search process to compute the locally optimal reduction directly. This proved fast for very small systems but it scaled poorly with the size of the system. The reason was that the advanced heuristic had to search through all potential paths in a much larger part of the search space, which with the current heuristic can be pruned away without any extensive search.

Given two subsystems $G_i = (Q_i, \Sigma_i, \rightarrow_i, q_{0i}, Q_{mi}, c_i)$ for $i \in [1, 2]$ and a specific synchronized state $\{\langle (q_1, t_1), (q_2, t_2)\rangle\}$ in $G_1 \parallel' G_2 = (Q, \Sigma, \rightarrow, q_0, Q_m, c)$. Let $\texttt{heuristic}\,(G_1, G_2, \langle (q_1, t_1), (q_2, t_2)\rangle)$ be a function that generates a set of outgoing transitions T from the current state following specific criteria. The generation of these transitions can be separated into three parts: the generation of *shared transitions*, *local parallel transitions* and *local single transitions*. The complete process of computing the heuristic is summarized in Algorithm 3 in the end of this section.

Shared transitions: If both G_1 and G_2 are ready to execute new events, i.e. $t_1 = t_2 = 0$, then, for all shared transitions $q_1 \xrightarrow{\langle \sigma, w_1\rangle} q_1' \in \rightarrow_1, q_2 \xrightarrow{\langle \sigma, w_2\rangle} q_2' \in \rightarrow_2$, add corresponding shared transition to T, given by

$$\langle (q_1, 0), (q_2, 0)\rangle \xrightarrow{\langle \sigma, \max(w_1, w_2)\rangle} \langle (q_1', 0), (q_2', 0)\rangle \in T. \tag{6}$$

The shared transitions are similar to regular synchronization, they requires both systems to execute simultaneously and to wait until the one with longest duration is completed, which creates a single transition where both systems takes a full step. Figure 8 shows an example where a shared transition, triggered by event e, is generated.

Local parallel transitions: If both G_1 and G_2 are ready to execute new events, i.e. $t_1 = t_2 = 0$, then all pairs of local transitions,

$$\left\{ q_1 \xrightarrow{\langle \sigma_1, w_1 \rangle} q_1', q_2 \xrightarrow{\langle \sigma_2, w_2 \rangle} q_2' \in \rightarrow_1 \times \rightarrow_2 | \sigma_1 \notin \Sigma_2 \wedge \sigma_2 \notin \Sigma_1 \right\}, \tag{7}$$

can be executed in parallel in the synchronized system using two sequential transitions . This is done by adding the first of the two transitions to the set T with the duration set to zero following Eq. 8. The second transition will then be added automatically as a local single transition in a later stage when the target of the first transition is explored.

$$\begin{aligned} \langle (q_1, 0), (q_2, 0) \rangle &\xrightarrow{\langle \sigma_1, 0 \rangle} \langle (q_1', w_1), (q_2, 0) \rangle \in T && w_1 \geq w_2 \\ \langle (q_1, 0), (q_2, 0) \rangle &\xrightarrow{\langle \sigma_2, 0 \rangle} \langle (q_1, 0), (q_2', w_2) \rangle \in T && w_1 < w_2 \end{aligned} \tag{8}$$

The criteria for generation of parallel transitions is more complex than that of the shared transitions. The upper path of Fig. 8c illustrates how two local parallel transitions are modelled where events a, b are executed immediately after each other (no delay in between). Event b is started at time zero (in the initial state) with zero duration. This allows event a to also start at time zero in the next state. Event a is set to execute for 5 seconds allowing the system to finish event a after 1 second and event b after 4 seconds. The reason that the event with the longest duration is started first is that this gives the algorithm the opportunity to execute multiple short local transitions in one system parallel to a long transition in the other system.

Local single transitions: There are three specific situations where a transition should be executed in one system while no transition is executed in the other. This is when the other system (i) continues an ongoing transition, (ii) waits in a marked state or (iii) waits for a shared transition. To clarify, if any of the aforementioned situations applies while G_1 is ready to execute a new transition, i.e. $t_1 = 0$, then each local single transition

$$\{ q_1 \xrightarrow{\langle \sigma_1, w_1 \rangle} q_1' \in \rightarrow_1 | \ \sigma_1 \notin \Sigma_2 \} \tag{9}$$

can be executed in G_1 without executing a transition in G_2. This is modelled according to the following criteria:

– if $t_2 \neq 0 \wedge w_1 \geq t_2$, then:

$$\langle (q_1, 0), (q_2, t_2) \rangle \xrightarrow{\langle \sigma_1, t_2 \rangle} \langle (q_1', w_1 - t_2), (q_2, 0) \rangle \in T; \tag{10}$$

– if $t_2 \neq 0 \wedge w_1 < t_2$, then:

$$\begin{aligned} \langle (q_1, 0), (q_2, t_2) \rangle &\xrightarrow{\langle \sigma_1, w_1 \rangle} \langle (q_1', 0), (q_2, t_2 - w_1) \rangle \in T \\ \langle (q_1, 0), (q_2, t_2) &\xrightarrow{\langle \sigma_1, t_2 \rangle} \langle (q_1', 0), (q_2, 0) \rangle \in T; \end{aligned} \tag{11}$$

– if $t_2 = 0 \wedge (q_2 \in Q_{m2} \vee \Sigma_2(q_2) \cap \Sigma_1 \neq \emptyset)$, then:

$$\langle (q_1, 0), (q_2, 0) \rangle \xrightarrow{\langle \sigma_1, w_1 \rangle} \langle (q_1', 0), (q_2, 0) \rangle \in T. \tag{12}$$

Note that two separate transitions are added in Eq. 11. The reason is that the synchronized system must have the alternative to either: (i) start a new local single transition in G_1 once the first transition is done, or (ii) to wait with the execution of any additional transitions until G_2 has completed its current task. The second alternative can be seen in the right path

of Fig. 8c, where a is executed in state $\langle(s_1, 0), (s_2, 5)\rangle$ with a time step of 5 seconds to allow G_2 to finish. The first alternative would in this case have been $\langle(s_1, 0), (s_2, 5)\rangle \xrightarrow{\langle a,1\rangle} \langle(s_2, 0), (s_2, 4)\rangle$. However, this was removed when trimming the final model, since it would have been a blocking state. Another example of a local single transition can be seen in the left path of the figure, where a is executed in G_1, while G_2 is waiting for the shared event e to be available. A side effect of the static property of the heuristic is that, in the second state of the path $(\langle(s_2, 0), (s_1, 0)\rangle)$, the heuristic does not remember that it had already neglected the b transition in that path so it will also include the option to run this transition in sequence with the first, which never can be an optimal solution. The criteria are analogue in the reverse situation when G_2 is to execute a transition while G_1 is waiting.

Algorithm 3 PTWS heuristic: Given two systems $G_i = (Q_i, \Sigma_i, \rightarrow_i, q_{0i}, Q_{mi}, c_i)$ for $i \in [1, 2]$ and a specific synchronized state $\langle(q_1, t_1), (q_2, t_2)\rangle$ in $G_1 \parallel' G_2 = (Q, \Sigma, \rightarrow, q_0, Q_m, c)$, compute the outgoing transitions from the the synchronized state.

1. Initiate the empty set $T := \{\}$ to become the set of outgoing transitions from state $\langle(q_1, t_1), (q_2, t_2)\rangle$.
2. If $t_1 = t_2 = 0$:
   - i For $q_1 \xrightarrow{\langle\sigma_1,w_1\rangle} q_1' \in \rightarrow_1$, $q_2 \xrightarrow{\langle\sigma_2,w_2\rangle} q_2' \in \rightarrow_2$:
     - If $\sigma_1 = \sigma_2$:
       Add a shared transition based on (6).
     - If $\sigma_1 \notin \Sigma_2 \wedge \sigma_2 \notin \Sigma_1$:
       Add one of the local parallel transitions given by (8).
   - ii If $q_2 \in Q_{m2} \vee \Sigma_2(q_2) \cap \Sigma_1 \neq \emptyset$:
     - For $\{q_1 \xrightarrow{\langle\sigma_1,w_1\rangle} q_1' \in \rightarrow_1 | \, \sigma_1 \notin \Sigma_2\}$:
       Add a local single transition based on (12).
   - iii Duplicate the process of step $(2.ii)$ while interchanging G_1 and G_2.
3. Else If $t_1 = 0$:
   - i For $\{q_1 \xrightarrow{\langle\sigma_1,w_1\rangle} q_1' \in \rightarrow_1 | \, \sigma_1 \notin \Sigma_2\}$:
     - Add local single transitions based on either (10) or (11), depending on the ratio between $w1$ and $t2$.
4. Duplicate the process of step (3) while interchanging G_1 and G_2.
5. Return T.

5.3 Implementation of PTWS

As previously mentioned, the implementation of PTWS is based on a simple forward search algorithm that adds new transitions based on the heuristic defined above. Pseudo code of the implementation is presented in Algorithm 4.

The algorithm starts by initiating an empty synchronous composition G, only including the initial state $\langle(q_{01}, 0), (q_{02}, 0)\rangle$. It then uses the heuristic defined above to compute the set of outgoing transitions from the initial state and adds these to G. For those transitions that lead to a previously unidentified state, this state is added to an ordered set S that acts as

a queue of states to expand next. The state is, of course, also added to the set of states Q in G and to the set of marked state Q_m when applicable.

The algorithm then successively expands G by evaluating the states in S one by one, performing the same steps. It terminates when there are no unexplored states left in S. In the end, the final model is trimmed to remove any blocking states that has been generated by the synchronization.

Algorithm 4 Partial Time-weighted synchronization: Given two weighted automatons $G_i = (Q_i, \Sigma_i, \rightarrow_i, q_{0i}, Q_{mi}, c_i), i \in \{1, 2\}$, compute the PTWS $G = G_1 \parallel' G_2$ of the two subsystems.

1. Initiate $G = (Q, \Sigma, \rightarrow, q_0, Q_m, c) :=$
   $(\{\langle(q_{01}, 0), (q_{02}, 0)\rangle\}, \Sigma_1 \cup \Sigma_2, \{\}, \langle(q_{01}, 0), (q_{02}, 0)\rangle, \{\}, c)$
2. If $q_{01} \in Q_{m1} \wedge q_{02} \in Q_{m2}$:
   $Q_m := Q_m \cup \{\langle(q_{01}, 0), (q_{02}, 0)\rangle\}$
3. $S := \{\langle(q_{01}, 0), (q_{02}, 0)\rangle\}$ is an ordered set of states to expand, where
   $S(i) : \langle(q_1, t_1), (q_2, t_2)\rangle$ is a function that returns the state at index i in S
4. $k := 1$
5. While $k \leq |S|$:
   i $\langle(q_1, t_1), (q_2, t_2)\rangle := S(k)$
   ii $T :=$ `heuristic` $(G_1, G_2, \langle(q_1, t_1), (q_2, t_2)\rangle)$
   iii $\rightarrow := \rightarrow \cup T$
   iv For $\langle(q_1, t_1), (q_2, t_2)\rangle \xrightarrow{\langle\sigma, w\rangle} \langle(q_1', t_1'), (q_2', t_2')\rangle \in T$,
   if $\langle(q_1', t_1'), (q_2', t_2')\rangle \notin S$:
      – $S := S \cup \{\langle(q_1', t_1'), (q_2', t_2')\rangle\}$
      – $Q := Q \cup \{\langle(q_1', t_1'), (q_2', t_2')\rangle\}$
      – If $t_1' = t_2' = 0 \wedge q_1' \in Q_{m1} \wedge q_2' \in Q_{m2}$:
        $Q_m := Q_m \cup \{\langle(q_1', t_1'), (q_2', t_2')\rangle\}$
   v $k := k + 1$
6. Trim the final model G
7. Return G

Example Consider the system of systems previously presented in Fig. 8. Applying Algorithm 4 on G_1, G_2 includes the following steps.

– *Initiation:* $G := (\{\langle(s_1, 0), (s_1, 0)\rangle\}, \{a, b, e\}, \{\}, \langle(s_1, 0), (s_1, 0)\rangle, \{\}, c)$, no changes to Q_m since $s_1 \in Q_{m1} \wedge s_1 \in Q_{m2} \neq True$, $S := \{\langle(s_1, 0), (s_1, 0)\rangle\}$.
– *Iteration 1:*
   i The first state to expand is the initial state $\langle(s_1, 0), (s_1, 0)\rangle$.
   ii $T :=$ `heuristic`$(G_1, G_2, \langle(s_1, 0), (s_1, 0)\rangle)$ will return the following two transitions: (i) no shared transitions, since no pair of transitions exists in the initial state with a shared event, (ii) in accordance with Eq. 8, the pair of local transitions $(s_1, 0) \xrightarrow{\langle a, 1\rangle} (s_2, 0) \in \rightarrow_1$ and $(s_1, 0) \xrightarrow{\langle b, 5\rangle} (s_2, 0) \in \rightarrow_2$ will give one local parallel transition $\langle(s_1, 0), (s_1, 0)\rangle \xrightarrow{\langle b, 0\rangle} \langle(s_1, 0), (s_2, 5)\rangle$, and (iii) since

$t_2 = 0 \wedge \Sigma_2(s_1) \cap \Sigma_1 = \{e\} \neq \emptyset$, the local single transition $\langle (s_1, 0), (s_1, 0)\rangle \xrightarrow{\langle a,1\rangle} \langle (s_2, 0), (s_1, 0)\rangle$ is included, in accordance with Eq. 12.

iii The transitions in T are added to $\rightarrow$.

iv The target states $\langle (s_1, 0), (s_2, 5)\rangle$, $\langle (s_2, 0), (s_1, 0)\rangle$ are added to both Q and S since neither of them has been previously explored. None of them are added to Q_m.

– *Iteration 2:*

i The second state to expand is $\langle (s_1, 0), (s_2, 5)\rangle$.

ii Since G_2 is busy executing the previous transition ($t_2 = 5$), the heuristic will in this case only have to consider local single transitions. This gives two transitions $\{\langle (s_1, 0), (s_2, 5)\rangle \xrightarrow{\langle a,1\rangle} \langle (s_2, 0), (s_2, 4)\rangle, \langle (s_1, 0), (s_2, 5)\rangle \xrightarrow{\langle a,5\rangle} \langle (s_2, 0), (s_2, 0)\rangle\}$, according to the criteria defined in Eq. 11.

iii The transitions in T are added to $\rightarrow$.

iv The target states $\langle (s_2, 0), (s_2, 4)\rangle$, $\langle (s_2, 0), (s_2, 0)\rangle$ are added to both Q and S since neither of them has been previously explored. The state $\langle (s_2, 0), (s_2, 0)\rangle$ is added to Q_m since $t_1' = t_2' = 0 \wedge s_2 \in Q_{m1} \wedge s_2 \in Q_{m2}$.

– *Iteration 3:*

i The third state to expand is $\langle (s_2, 0), (s_1, 0)\rangle$.

ii In this case, the heuristic gives a total of two transitions: (i) one shared transition $\langle (s_2, 0), (s_1, 0)\rangle \xrightarrow{\langle e,2\rangle} \langle (s_2, 0), (s_2, 0)\rangle$, (ii) no local parallel transitions, since no local transitions are available in G_1, i.e. $\Sigma_1^l(s_2) = \emptyset$, and finally (iii) one local single transition $\langle (s_2, 0), (s_1, 0)\rangle \xrightarrow{\langle b,5\rangle} \langle (s_2, 0), (s_2, 0)\rangle$, since $t_1 = 0 \wedge s_2 \in Q_{m1}$.

iii The transitions in T are added to $\rightarrow$.

iv Both transitions in T have the same target state $\langle (s_2, 0), (s_2, 0)\rangle$. This state has been seen before and is already part of the set S.

– *Iteration 4 and 5:*
There are two remaining states in S: $\langle (s_2, 0), (s_2, 4)\rangle$ and $\langle (s_2, 0), (s_2, 0)\rangle$. Neither of these will have any outgoing transitions since G_2 does not have any available transitions and the only transition in G_1 is a shared transition.

– *Termination:*
There are no further states in S to explore. Trimming G will remove transition $\langle (s_1, 0), (s_2, 5)\rangle \xrightarrow{\langle a,1\rangle} \langle (s_2, 0), (s_2, 4)\rangle$ and its target state $\langle (s_2, 0), (s_2, 4)\rangle$, since this state is blocking, i.e. it is impossible to reach a marked state from here.

Computational complexity: The main benefit with a static heuristic is, as previously explained, its low impact on the computational complexity. This is especially true for the *memory complexity*, which refers to how the memory allocation of the algorithm scales with the number of states in the system. It is shown later in Section 6 that the computation time of the optimization can be directly correlated to the total memory allocated by the algorithm. The heuristic requires no additional memory allocation, since it only uses information about outgoing transitions of the current states of the subsystems, which can be retrieved directly from the system models.

To calculate the theoretically worst case computational complexity of the synchronization in Algorithm 4, one should look at the expansion of states in the synchronous composition. Following the enumeration of Algorithm 4 one can see that (5) considers each

state in S, $\mathcal{O}(|S|)$. For each such state the heuristic function is computed in (5.ii), where the heuristic function considers each pair of outgoing transitions from the current states, $\mathcal{O}(|\rightarrow_1 \times \rightarrow_2|)$. In (5.iv) the transitions generated by the heuristic is analyzed again to generate new states which only adds a multiplier to the complexity of the heuristic. However, the generation of new states is the key to the total complexity, since this drives the size of the explored state space S. Due to the the properties of the heuristic, the states in S follows one of the following criteria: (i) both of the subsystems is in a state (i.e $t_1 = 0 = t_2$), (ii) subsystem G_1 is in a state while G_2 is currently executing a transition (i.e $t_1 = 0 \neq t_2$), (iii) subsystem G_2 is in a state while G_1 is currently executing a transition (i.e $t_1 \neq 0 = t_2$). Hence the total amount of states that can be generated from the heuristic during a full synchronization is $|Q_1 \times Q_2| + |Q_1 \times \rightarrow_2| + |Q_2 \times \rightarrow_1|$.

Let $V = \max(|Q_1|, |Q_2|)$ and $E = \max(|\rightarrow_1|, |\rightarrow_2|)$ be the maximum number of states and transitions respectively, where $E >> V$ in general. From this we can formulate the worst case computational complexity of Algorithm 4 as

$$\mathcal{O}(|V \times E||E \times E|) = \mathcal{O}(VE^3). \tag{13}$$

6 Application

This section illustrates the actual complexity and potential of the algorithms when optimizing a system of systems that have subsystems with complex local behavior. This is done in a series of experiments using a simplification of a respotting problem in a welding robot cell. Just like the real scenario, the example includes multiple robots that operate in parallel on the same product but from different angles. During a production cycle there are specific disruptive events that affect all robots similarly, such as the assembly of one additional sub-part to the product. A few of the welding operations performed by the robots has to be performed while the assembly robot is still gripping the part, but a majority of the operations can or has to be performed once the assembly robot has left the zone.

These disruptive events typically put strict constraints on most tasks in the cell to be performed either before or after the global event. This constitutes the dependencies between the systems and makes it impossible to only optimize each robot individually without risking ending up with a sub-optimal solution. This is a common scenario in applications where the subsystems mainly work independently but have to collaborate regarding some global or shared behavior. It is also just the type of structure that can be exploited by the CompOpt method, since most of the behavior in each subsystem is local.

The goal of the optimization is to find a globally optimal schedule for one repeated cycle of the robot cell. This result can be represented as a Gantt chart such as Fig. 9, which shows a globally optimal schedule for a specific instance of the given example.

The size of the example can be modified in multiple ways to evaluate how the actual complexity of the algorithms scales with different properties of the problem instance. The results shown in this section focus on how the size of the reduced state space scales when the system grows. This is also compared to time and memory allocation of the computation to verify the complexity of the algorithms used in the optimization.

6.1 Modelling of the example

The example is based on a robot cell that includes a set of robots $R = \{r_1, \ldots, r_n\}$ that represent the subsystems of the plant. Each robot $r_i \in R$ can operate in an independent

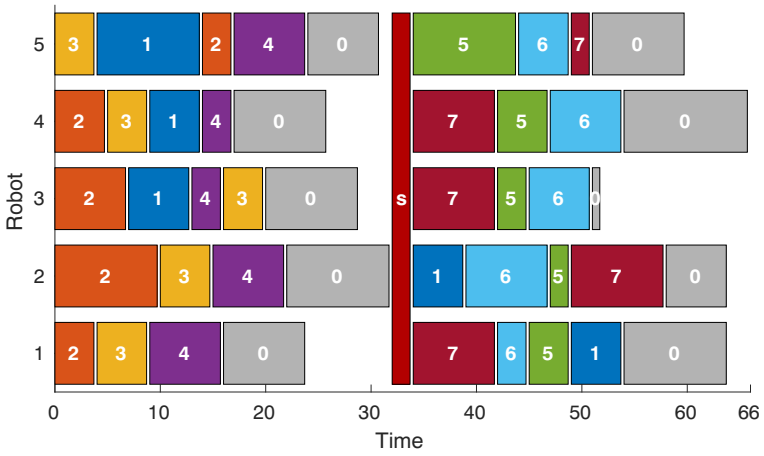


Fig. 9 The globally optimal schedule for a specific instance with 5 robots and 7 tasks per robot. Block j on robot i represents the travel to and execution of task t_{ij}. Event 0 represents the action when a robot returns to its initial state and s represents the global event

area A_i of size $x_{max} \times y_{max}$. Recent work by Åblad et al. (2017) shows that it is possible to generate (more complex) zones such that collisions between robots are guaranteed to be avoided. In a single cycle of the cell, each robot r_i is required, from a home position or idle state, to go to and perform m tasks of duration d that are located at random locations in A_i. The tasks are pair-wise independent and can be performed in arbitrary order. When all tasks are finished, each robot should return to its idle state. In addition to the individual tasks there is also one global event s with duration d_s that simulates a disruptive change of the product which affects all robots.

Specific problem instances for the example are represented by a set of automata $\mathbf{G} = \{G_i\}, \forall r_i \in R$. Each element $G_i \in \mathbf{G}$ models the full behavior of a robot including the tasks to be performed. The process of generating these instances are now presented.

Selecting coordinates for each task: For each robot $r_i \in R$, m coordinates (x_{ij}, y_{ij}) should be randomly selected to represent the position within A_i of the task t_{ij}, where i, j represent the index of the robot and task respectively. This has been done using a random permutation P_i of the integers $[1, x_{max}\, y_{max}]$ for each robot. The coordinates are then selected as $x_{ij} = (p_{ij} - 1) \mod x_{max} + 1$ and $y_{ij} = \lceil p_{ij}/x_{max} \rceil$, where $a \mod b$ is the non-negative remainder when dividing a by b. This gives $x_{max} \times y_{max}$ possible locations for the tasks of each robot and ensures that no two tasks are in the same position. The random permutation is generated using Mersenne Twister as random number generator, accepting a specific seed to be used to create an unlimited number of reproducible instances (Matsumoto and Nishimura 1998).

Modelling of the individual robots: The behavior of each robot r_i is modelled as a weighted automaton including the physical movement and the execution of the global and individual tasks. The model is restricted to only one initial state q_0 representing the idle state of the robot and one state q_j for each task t_{ij}. The idle state is also a marked state. Between all pair of states $q, q' \in Q$ there is a transition $q \xrightarrow{\langle \sigma, w \rangle} q'$, where σ is a local event σ_{ij} corresponding to the movement and execution of task j and w represent the cost $c(q_j, \sigma_{ij})$

given by the Eucledian distance between the tasks plus the duration d (the idle state is set to be located in $(x_i^i, y_i^i) = (0, 0)$ and there is no duration added when entering this state). This means that all transitions that share the same target also share the same event but with different weights. In addition to this, there exists a self-loop $q_0 \xrightarrow{\langle s, d_s \rangle} q_0$ simulating that the robot first needs to return to the idle state before the global event can occur. Figure 10a shows an example of a robot model for an instance with three tasks per robot.

Ensure that each task is performed exactly once: This is done by including one individual automata for each task that requires the task to be performed before entering a terminating marked state, see Fig. 10b.

Construct specifications with regards to the global event: To simulate the desired global behavior, specific precedence constraints are added between the global event and most individual tasks. These specify whether the task should precede or be preceded by the global event. This is modelled by automata such as Fig. 10c.

The example could in principle be varied with all possible combination of precedence constraints but some simplifications has been made when generating problem instances to make the evaluation of the complexity more accurate. For this reason, each robot has been constrained in the same way to better isolate the different behaviors. However, since the tasks themselves are randomly generated the behavior still differs between similar instances. The m tasks in each robot have been divided as follows: the first k tasks are considered independent of the global event, the following $\lceil (m-k)/2 \rceil$ tasks have to precede the global event and the rest have to be preceded by the global event. For example, if $m = 4$ and $k = 1$ then t_{i1} will be independent, t_{i2}, t_{i3} will precede and t_{i4} will be preceded by the global event for each robot $r_i \in R$.

Combining the parts into separate subsystems: The last step when generating a problem instance is to synchronize all models described above into one subsystem Gr_i that represents the complete behavior of each robot. Synchronization of these components does not require PTWS, since each robot only can perform one task at a time.

6.2 Evaluation of actual complexity

The evaluation has been done by solving multiple problem instances with different properties. The properties that has been varied are mainly the number of robots in the cell and the

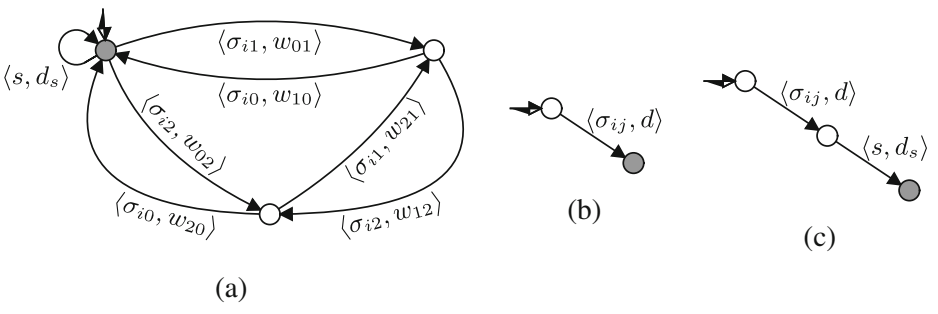


Fig. 10 Example of the individual models that constitutes a subsystem. **a** example of a robot model with two tasks, **b** model of each task t_{ij} ensures exactly one execution, **c** example of a preceding constraint where task t_{ij} is to precede the global event s

number of tasks performed by each robot, to test how the complexity scales with the amount of subsystems and complexity of their local behavior respectively. The results focus on the total number of states in the optimization of the system, i.e. the sum of the number of states in all sub-problems solved.

How the complexity scales with the number of subsystems is, as mentioned in Section 4.2, strongly connected with the complexity of the shared behavior. To show the potential of the method, the evaluation focus mainly on problem instances with a single independent task in each robot. In the end of this section we show how the method scales when the complexity of the global behavior increases to give an indication of the limitations of the method.

Scaling of state space size : The plots in Fig. 11 show how the total number of states in the optimization increase when increasing the number of subsystems (additional robots) or the complexity of their local behavior (more tasks per robot). This refers to the sum of the number of states in all sub-problems solved during the CompOpt, which represents the extent of the total search space. The complexity is evaluated for a varying number of robots ranging from 2-15 and tasks per robot ranging from 3-10. Note that the scale of the y-axis in the plots is logarithmic.

The system scales very differently for the two parameters. When only scaling the complexity of the local behavior the number of states scales almost linearly, the reason is that it only adds to the complexity of the initial optimization of the subsystems, which have a very limited effect on the total complexity since they are solved independently before synchronization. This is the main reason why CompOpt can be so efficient.

On the other hand, the number of states scales exponentially when increasing the number of robots in the system. The reason is that, even if the CompOpt decreases the size of each subsystem, there will still remain some shared behavior that cannot be solved until all subsystems have been synchronized. This will cause the state space to grow exponentially with the size of the shared behavior until the final optimization. However, this still scales very efficiently compared to a monolithic model.

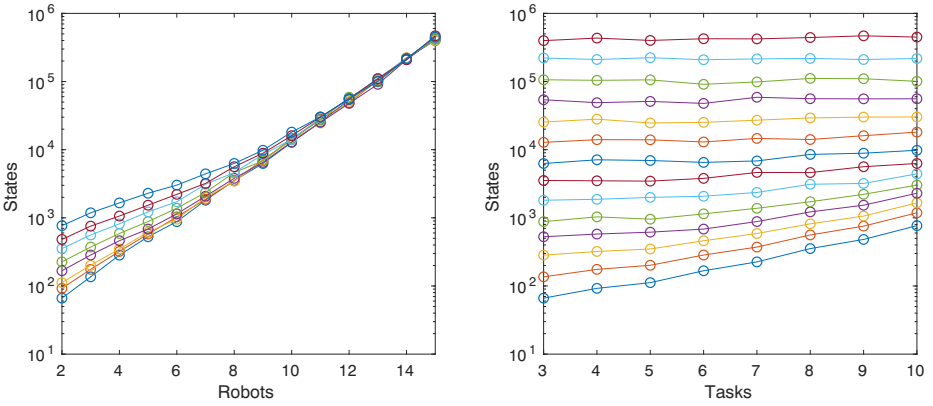


Fig. 11 Illustration of how the state space scales with the number of robots and tasks per robot. Each line in the plots represents a fixed number of robots in **a** ranging from 2-15 and a fixed number of tasks per robot in **b** ranging from 3-10

Comparison with a monolithic method: Table 1 shows how the state space of a trim monolithic model grows with an increased number of subsystems and/or an increased number of tasks per robot. This gives a hint on the size of the problem instances used and the efficiency of the CompOpt.

The extent of the table is limited by to the fact that the computation either timed out or ran out of memory when solving for any larger instances (apart from instances with only one robot which was considered irrelevant for the application). This is in contrast to the results below where systems with 10 robots and 10 tasks per robot were solved.

Looking at how fast the state-space explodes gives an indication why a modular or compositional approach is to prefer to a monolithic method.

Comparing with theoretical results: In Section 4.1 we show that the theoretical worst case complexity of the local optimization is polynomial for the number of states. To verify these results and to indicate the speed of the polynomial scaling, the time and memory allocation have been measured during each optimization. Computation time has also been calculated as an average of 10 independent executions for identical instances. The measurements are shown in Fig. 12.

The plots in Fig. 12 also include a curve fit, where a second order polynomial has been fitted to the data. One can see that the quadratic curve follows the average growth of the data quite well both for time and space complexity, even if the quadratic function does not represent the data exactly. This indicates that the complexity is approximately $\mathcal{O}(n^2)$ for this specific example where n is the sum of the number of states in all sub-problems. However, as described in Section 4.2, this will generally not be true for a system with more extensive shared behavior between the subsystems.

Increasing the complexity of the global behavior: This section shows results that can give an indication of the dependencies between the efficiency of CompOpt and the complexity of the shared behavior in the subsystems. It is illustrated by a set of experiments using the same example as previously but increasing the number of independent tasks in each robot, that is how many tasks that are independent of the global event. These tasks can be performed either before or after the global event, which constitutes an alternative in the global behaviour of the subsystem and increases the global complexity of the system.

Table 1 Average number of states in the monolithic model for an instance with n robots with m tasks each, calculated using instances with ten different random seeds

m	n=1	n=2	n=3	n=4
2	117.0	8,140.0	657,440.0	35,961,344.0
3	188.6	15,392.0	1,379,880.8	-
4	420.6	96,030.8	27,150,565.3	-
5	627.6	180,032.0	-	-
6	1,263.6	912,585.6	-	-
7	1,882.4	1,638,088.0	-	-
8	3,503.2	6,986,696.0	-	-
9	5,098.4	11,981,064.0	-	-
10	9,237.6	42,450,952.0	-	-

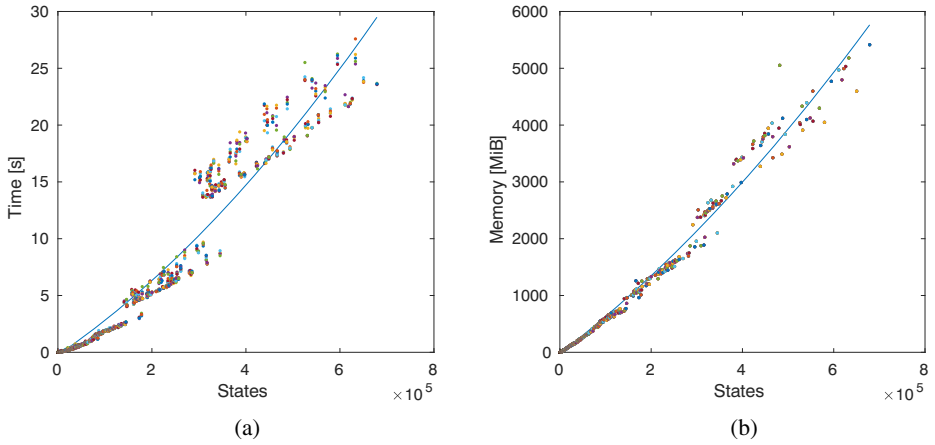


Fig. 12 Evaluation of how computational complexity scales with the number of states, measured as **a** computation time, **b** memory allocation

The evaluation uses a fixed number of 8 tasks per robot while varying the number of robots, since it already has been shown that this has a worse effect on the complexity. The results can be seen in Fig. 13 which show a significant increase of the complexity when increasing the number of independent tasks. An interesting note is that the average search space of when optimizing 2 robots with 6 independent tasks each has a size of approximately $4e^4$, which is less than 1% of the monolithic model of 2 robots with only 1 independent task each, shown Table 1.

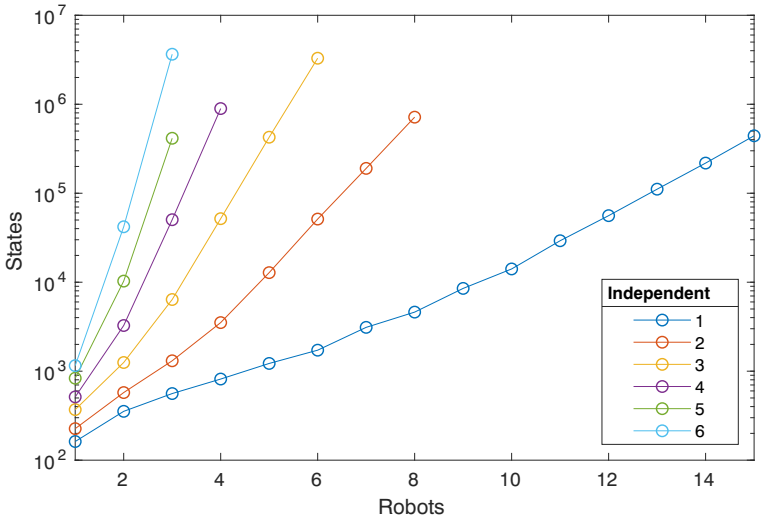


Fig. 13 Illustration of how the complexity of the example scales with the number of independent tasks

6.3 Comparison with previous work

All results presented in Section 6.2 have been calculated using the exact same instances of the example that were used in previous work. This allows for a comparison between the new synchronization method proposed in this paper with the old method, which used tick automata to model time-weighted systems. We can conclude that the results presented in this paper is approximately a magnitude of ten better than with the previous method, i.e. fewer states in the evaluations. This shows that CompOpt manages to optimize systems of larger scale when using PTWS due to the reduction in memory allocation.

However, the major improvement of PTWS becomes clear first when we generate instances where the duration or precision of the weights are increased. This will have a big negative effect to the state space when using the old method with tick automata, due to the discretization of the state space. So far the examples have used only small weights on the transitions and the tick automata has been allowed to round this up to nearest integer. This

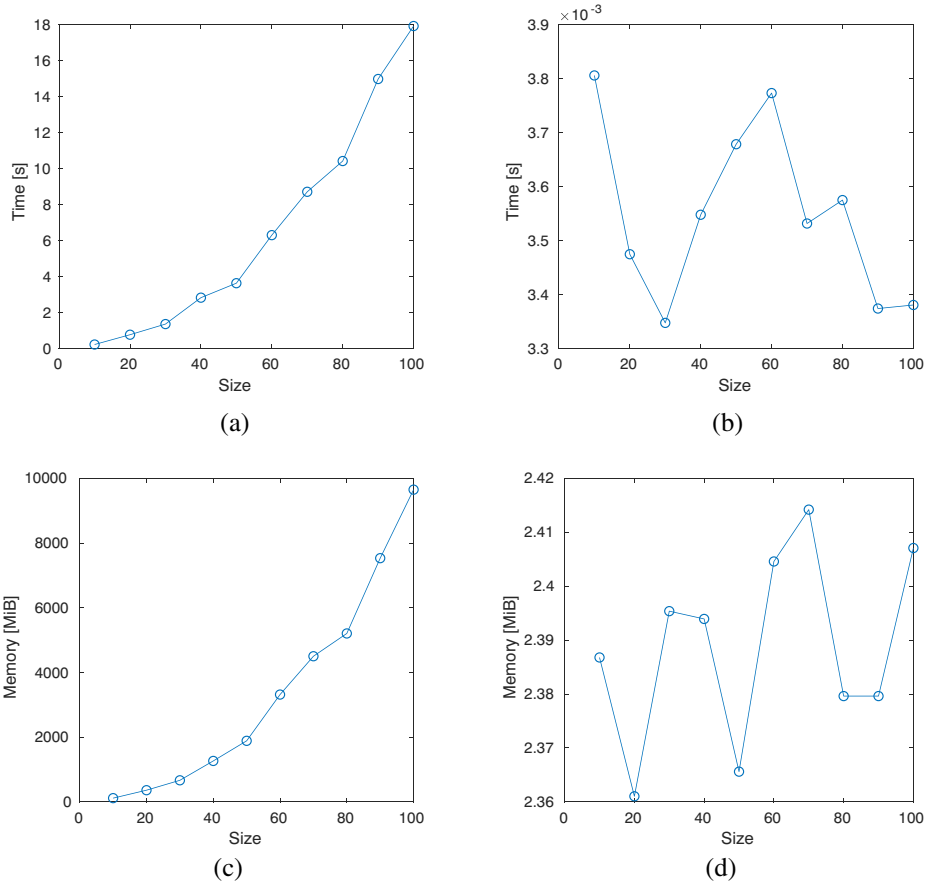


Fig. 14 Evaluation of how computational complexity scales with the size of the area of operation: **a** and **b** computation time using the old method and PTWS respectively, **c** and **d** memory allocation using the old method and PTWS respectively

is not realistic in industrial applications where the scheduling usually requires high precision. This can be achieved by letting a tick-event in the automata represent a shorter time unit, e.g. milliseconds, but this would drastically increase the number of states added to the model.

To evaluate the effect of an increased precision we have run a series of experiments where the number of robots and the number of operations per robot has been kept constant at 5 respectively, while increasing the size of the area of operation in each robot, i.e. changing the values of x_{max}, y_{max} mentioned in Section 6.1. This will automatically increase the average duration of the operations, since they mainly comes from the movement of the robot. The area of operation has been grown from the nominal value of 10×10 used by the experiments in Section 6.2 to 100×100. The results are presented in Fig. 14 where both time and memory complexity are compared. Note that the scale on the y-axis differs between the four graphs.

In Fig. 14 we see that the complexity of the old method increases rapidly when increasing the size of the area of operation while the complexity of the new method is mostly indifferent to changes to the size. The variance that still exists in the results using the new method is assumed to be caused by the variance in the individual problem instances, caused by the randomized generation of tasks, rather than a growth in complexity.

7 Conclusion

This paper continues the development of *compositional optimization*, an optimization method which previously has been proposed for large-scale systems of systems. The key to this method is a local optimization technique that reduces the size of each subsystem individually to mitigate the state explosion problem. It is proven that this local optimization maintains the global optimal solution of the system while removing all non-optimal or redundant paths. The method offers further reduction of the search space using a compositional algorithm that performs local optimization iteratively while synchronizing the subsystems. Moreover, dividing the problem into multiple sub-problems makes parallelization of the computation possible. This improves the applicability of the method significantly, since additional computation power can be purchased on-demand through cloud services. Results in this paper show that the method has the potential to scale very well with the number of subsystems. This is especially true with subsystems that have complex local behavior, something that in a monolithic optimization would cause an exponential growth of the search space. This makes it possible to calculate global optimal solutions for large-scale industrial applications.

A novel contribution in this paper is the introduction of a new synchronization method, called *partial time-weighted synchronization* (PTWS), that is specifically designed for a class of systems called time-weighted systems. The method further mitigates the state explosion problem by integrating an optimization heuristic into the synchronization that generates a reduced synchronous composition where many non-optimal or redundant solutions already have been removed. The synchronization of time-weighted systems were in previous work identified as one of the main limitations to compositional optimization and, yet, a majority of the current industrial applications are of this class. In this paper we show that the addition of PTWS greatly improves on this limitation. For these reasons, PTWS is considered to be a major improvement to compositional optimization.

In future work it would be of interest to implement parallel computation of compositional optimization as a cloud service to evaluate the potential of having scalable computation

power. Additionally, it would be interesting to apply this method as an online optimization method in real industrial applications.

Funding Information Open access funding provided by Chalmers University of Technology.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

Åblad E, Spensieri D, Bohlin R, Carlson JS (2017) Intersection-free geometrical partitioning of multirobot stations for cycle time optimization. IEEE Trans Autom Sci Eng PP(99):1–10

Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235. https://doi.org/10.1016/0304-3975(94)90010-8

Bertsekas DP, Tsitsiklis JN (1996) Neuro-dynamic programming. Athena Scientific, Belmont

Bertsekas DP (2005) Dynamic Programming and Optimal Control, vol I, 3rd edn. Athena Scientific, Belmont

Brandin BA, Wonham WM (1994) Supervisory control of timed discrete-event systems. IEEE Trans Autom Control 39(2):329–342

Cao X (2007) Stochastic Learning and Optimization: A Sensitivity-Based Approach. Springer, Berlin

Cassandras CG, Lafortune S (2008) Introduction to Discrete Event Systems, 2nd edn. Springer Science & Business Media, Berlin

David R, Alla H (2010) Discrete, Continuous, and Hybrid Petri Nets. Springer, Berlin. https://doi.org/10.1007/978-3-642-10669-9

Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1(1):269–271. https://doi.org/10.1007/BF01386390

Flordal H, Malik R (2009) Compositional verification in supervisory control. SIAM J Control Optim 48(3):1914–1938

Gass SI, Fu MC (2013) Encyclopedia of Operations Research and Management Science, 2013rd edn. Springer, Berlin

Gruber H, Holzer M, Kiehn A, König B (2005) On timed automata with discrete time – structural and language theoretical characterization. In: De felice C, Restivo A (eds) Developments in Language Theory. Springer, Berlin, pp 272–283

Hagebring F, Wigström O, Lennartson B, Ware SI, Su R (2016) Comparing MILP, CP, and A* for multiple stacker crane scheduling. In: 13th International Workshop on Discrete Event Systems (WODES), pp 63–70

Hagebring F, Lennartson B (2018) Compositional optimization of discrete event systems. In: 14th IEEE International Conference on Automation Science and Engineering

Hill R, Lafortune S (2016) Planning under abstraction within a supervisory control context. In: 2016 IEEE 55th Conference on Decision and Control (CDC)

Hill R, Lafortune s (2017) Scaling the formal synthesis of supervisory control software for multiple robot systems. In: 2017 American Control Conference (ACC)

Hoare C (1978) Communicating Sequential Processes, vol 21. ACM, New York. https://doi.org/10.1145/359576.359585

Huang J, Kumar R (2008) Optimal nonblocking directed control of discrete event systems. IEEE Trans Autom Control 53(7):1592–1603

Kobetski A, Fabian M (2009) Time-optimal coordination of flexible manufacturing systems using deterministic finite automata and mixed integer linear programming. Discret Event Dyn Syst 19(3):287–315

Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans Model Comput Simul 8(1):3–30

Mohajerani S, Malik R, Fabian M (2014) A framework for compositional synthesis of modular nonblocking supervisors. IEEE Trans Autom Control 59(1):150–162

Passino KM, Antsaklis PJ (1989) On the optimal control of discrete event systems. In: Proceedings of the 28th IEEE Conference on Decision and Control

Powell WB (2007) Approximate dynamic programming: Solving the curses of dimensionality. Wiley-Interscience, New York

Ramadge PJ, Wonham WM (1987) Supervisory control of a class of discrete event processes. SIAM J Control Optim 25(1):206–230
Ramadge PJ, Wonham WM (1989) The control of discrete event systems. Proc IEEE 77(1):81–98
Su R (2012a) Abstraction-based synthesis of timed supervisors for time-weighted systems. IFAC Proc Vol 45(29):128–134. https://doi.org/10.3182/20121003-3-MX-4033.00024
Su R, van Schuppen JH, Rooda JE (2012b) The synthesis of time optimal supervisors by using heaps-of-pieces. IEEE Trans Autom Control 57(1):105–118. https://doi.org/10.1109/TAC.2011.2157391
Valmari A (1998) The state explosion problem. In: Lectures on petri nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets
Ware S, Su R (2017) Time optimal synthesis based upon sequential abstraction and its application to cluster tools. IEEE Trans Autom Sci Eng 14(2):772–784
Wong KC, Wonham WM (1998) Modular control and coordination of discrete-event systems. Discret Event Dyn Syst 8(3):247–297

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Fredrik Hagebring was born in Borås, Sweden, in 1985. He received a M.Sc. degree in Systems, Control and Mechatronics from Chalmers University of Technology, Gothenburg, Sweden, in 2016. Since then, he has been pursuing a Ph.D. degree at Electrical Engineering Department, Chalmers.

Bengt Lennartson was born in Gnosjö, Sweden, in 1956. He received the Ph.D. degree from Chalmers University of Technology, Gothenburg, Sweden, in 1986. Since 1999, he has been a Professor of the Chair of Automation, Department of Electrical Engineering, and now he is Head of the Division of Systems and Control. From 2004 to 2007 he was Dean of Education at Chalmers University of Technology, and since 2005 he is also part-time Professor at University West, Trollhättan. Lennartson is IEEE Fellow for his contributions to hybrid and discrete event systems for automation and sustainable production. He was General Chair of the 11th IEEE Conference on Automation Science and Engineering, CASE 2015, and the 9th International Workshop on Discrete Event Systems, WODES08, and Associate Editor for Automatica 2002-2005 and IEEE Transaction on Automation Science and Engineering 2012-2015. He is (co)author of two books and 300+ peer reviewed papers in international journals and conferences. His main areas of interest include discrete event and hybrid systems, AI planning and learning, as well as robust feedback control.